ARMY RESEARCH LABORATORY

Dynamic Matrices and Their Application to the Concept of a Semi-Sparse Matrix

Daniel M. Pressel

ARL-MR-198 November 1994

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

19941208 012

NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1994	Final, Jan 87–May 94

4. TITLE AND SUBTITLE

Dynamic Matrices and Their Application to the Concept of a Semi-Sparse Matrix

5. FUNDING NUMBERS

Unfunded

6. AUTHOR(S)

Daniel M. Pressel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-CI-CA
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

ARL-MR-198

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report primarily describes research performed by the author while he was still employed by the U.S. Army Chemical Research, Development and Engineering Center, as well as continuing research being conducted by the author at the U.S. Army Research Laboratory, Aberdeen, MD.

In a number of cases (e.g., tridiagonal matrices, triangular matrices, and the production of raster images from wireframe CAD data), one is left with the problem of efficiently storing and accessing matrices which are large enough to worry about, and have regions of a meaningful size with no data (or if one prefers, some default value, usually zero), and other regions with at least some data. In many cases, these matrices contain too much data to be efficiently handled using the linked lists commonly used with sparse matrices. On the other hand, they may be too large and too sparse to be efficiently handled using normal methods. The author has named the matrices which fall into this category, SEMI-SPARSE MATRICES. While it is true that there are well-known application-specific methods for dealing with some types of matrices in this category (e.g., triangular matrices), there appears to be no standard method for dealing with this category as a whole. In response to some of the problems this caused, the author has developed and implemented the concept which he refers to as a DYNAMIC MATRIX.

14. SUBJECT TERMS

computer programming, matrices, sparse matrix

15. NUMBER OF PAGES
53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

ACKNOWLEDGMENTS

The author wishes to thank his colleagues at the Computational Sciences and Technology Division of the Advanced Computational and Information Sciences Directorate, U. S. Army Research Laboratory, for their help in the preparation of this report. Additionally, the author would like to thank the Edgewood Research, Development and Engineering Center of the Chemical Biological Defense Command for providing the impetus for the work this report was originally based on. Finally, a special thanks goes to John Petresky and Kathy Burke of the U.S. Army High Performance Computing Research Center/Computer Sciences Corporation for their valued assistance in bringing this report to fruition.

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

Accesion For
NTIS CRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

INTENTIONALLY LEFT BLANK.

1. INTRODUCTION

Traditionally, matrices have been grouped into three categories—sparse, regular, and special cases (e.g., triangular matrices). While it is difficult to give a definition of what a sparse matrix is, one might agree that any matrix in which significantly less than 1% of the elements have a nonzero value should be considered to be sparse. It is traditional to represent sparse matrices using linked lists. When considering the efficiency of this representation, there are several aspects which should be considered. For example, How long will it take to access an element? or How much space will be required to store the matrix? As the dimensions of the matrix become large, it may no longer be practical to use the traditional techniques (especially if the size of the data element is very small).

Sometimes matrices are neither dense nor truly sparse (e.g., 5% of the space contains nonzero values). If sparse matrix techniques are applied to matrices in this category, there may be little or no saving of space. Even worse, the order N access time (written O(N)) for this representation (where N is the larger dimension of the matrix) is likely to so affect the speed of the program as to make it impractical to perform a significant number of operations on multimillion element matrices.† On the other hand, attempting to treat such a matrix in the normal fashion (e.g., row major or column major) may require so much space as to either exceed the available amount of memory, or in a virtual memory environment, cause the computer to thrash to death. The author is not the first person to recognize this problem, and the literature is full of specialized techniques for handling a number of special cases (e.g., triangular matrices). Unfortunately, none of these techniques seem to be general in nature, and in some cases it can be difficult to recognize the matrix structure.

---

† For those unfamiliar with this terminology, an algorithm is referred to as having order 1 execution time if it takes a constant amount of time to execute regardless of the size of the data set. This is frequently written as O(1). Similarly, an algorithm whose execution time scales linearly with the size of the data set is said to have order N execution time. Again, this would be written as O(N). In some cases, one must carefully define what N is, since an algorithm which adds a constant to each element of a square matrix is O(N) in terms of the total number of elements in the matrix, but $O(N^2)$ if N is the number of rows in the matrix. The basis for this terminology is that many algorithms have an execution time which can be bounded by a polynomial in some suitably selected value of N. If all of the coefficients of this polynomial have roughly the same value (e.g., to within a factor of 100), then as N becomes large, the highest order term in the polynomial will dominate, and this power is considered to be the order of the execution time. In reality, one sometimes will find that when using specific hardware, it is impractical to let N become large (e.g., one runs out of memory), or that as N becomes large, the values of some of the coefficients will change (e.g., the memory access time suddenly becomes dominated by paging activity). The author believes that areas such as this present opportunities for research into new solutions to what are generally considered to be solved problems, and, that without this research, one may unnecessarily conclude that one needs substantially more powerful hardware to perform a specific task than may in fact be needed. When applied to the field of supercomputers, this may be a key factor in determining which algorithms can run on which computers. One final note, while this footnote has been discussing polynomial time orders (e.g., $O(N^2)$), it is also possible for an algorithm to have a nonpolynomial time order (e.g., O(N log N), 2^N, $N^{2.61}$, etc.).

In response to these and other concerns, the author started looking for alternative ways to store matrices and came to the following conclusions.

- In order to be practical, the access time should be O(1) (all elements can be accessed in the same amount of time). This is the same order of complexity as the standard method for accessing matrices.

- Ideally, the method should have a sufficiently low amount of memory overhead as to be useful for storing both sparse matrices, as well as matrices with nonzero data in as much as 25% of the elements (and in some cases 50% or more of the elements).

- It is acceptable for the new technique to have a longer access time, so long as the increase is not excessive (noting that this is a somewhat subjective criterion). In particular, the increase in time should be significantly less than a factor of N (defined as before) for values of N in the range of interest.

Looking for a unifying concept on which to base the new form of matrix representation, the author realized that almost all matrices which have a large percentage of elements with the value of zero tend to share one property in common; they are likely to have significant regions which are totally devoid of nonzero data. To use a metaphor, the matrix consists of "islands of data in a sea of zeros." The author refers to matrices which have this property as being "semi-sparse."

Having found the unifying concept, it is possible to construct a data structure which, roughly speaking, allocates storage space only for those regions of the matrix which actually contain nonzero data. It turns out that this is simpler said than done, but that if one is willing to allocate space for a small percentage of the zeroed out elements, then the task becomes doable. The author has named his implementation of this concept a "dynamic matrix."

Finally, this report will also consider some other potentially useful data structures, with the goal of defining what might be called an "extensible dynamic matrix." While the author has not yet implemented this concept, its purpose would be to allow the development of more robust applications, which to a first approximation would allocate arrays and matrices of exactly the size needed without explicitly using pointers.

2. A BRIEF REVIEW OF TRADITIONAL TECHNIQUES FOR STORING MATRICES

For languages which directly support the concept of a matrix, the most common representations of a matrix involve storing either the first column or row of the matrix in consecutive addresses, followed by the next column or row, respectively, and so on (this assumes that one is talking about two-dimensional matrices, but is easily extended to multidimensional arrays). In general, this is considered to provide the most efficient representation of a matrix since it is both compact and the address of each element can be easily calculated (meaning each element can be efficiently accessed in a fixed amount of time).

Almost as soon as these representations were developed, it was realized that there were cases in which the vast majority of the elements had the value zero. This class of matrices became known as "sparse matrices," and the search was on for an efficient way to store these matrices. The most common solution to this problem involves the use of linked lists (frequently bidirectionally linked in all dimensions). This technique had the advantage that the amount of space required to store the matrix is equal to A * N + B * M, where A and B are small positive integers, N is the largest dimension of the matrix, and M is the number of nonzero elements in the matrix. Unfortunately, it is no longer possible to compute the address of an element in the matrix. Instead, one must follow the linked lists, and, if N is large, this can be rather time-consuming. Moreover, the amount of storage space required for pointers may not be small, so that as the value of M approaches the total number of elements in the matrix, this representation will eventually take more space than the normal method for storing matrices. In particular, this means that this method is ill-suited for storing triangular matrices.

As a result of these considerations, a substantial amount of work has gone into developing more efficient methods for dealing with a number of special cases (e.g., triangular and tridiagonal matrices). While, in general, these techniques make very efficient use of space, and can be computationally efficient (although frequently somewhat less efficient than the normal method for storing matrices), by their very nature they lack generality. As a result, it is necessary to write special versions of routines which manipulate matrices to handle these special methods of representation. Since for moderate sized problems, the programmer's time can be more valuable than the computer time, this raises the question of how useful these methods really are. (Note, that if the program is run often enough, the answer can still be that they are very useful.)

3. HOW MATRICES ARE USED

Before continuing on, it might be helpful to briefly discuss how matrices are used. Probably the most common use is to store information, which will then be transformed using common techniques such as matrix multiplication. Computer Aided Design (CAD) systems routinely use them to store geometry data, with matrix multiplication used to perform rotations, translations, and projections. Another application which might come to mind is the representation of a set of simultaneous linear equations in a manner which facilitates the use of Gaussian elimination.

Computer scientists have also learned that matrices are a useful method for storing a wide range of other types of information, each with their own set of operations. Some common examples are the creation and manipulation of raster images (e.g., transforming data from a CAD system into a format compatible with a dot matrix plotter), the accumulation of data about each point on a grid over a fixed period of time (e.g., the amount of rain which has fallen at that grid point might be updated every 5 minutes), or storing the numeric values for a spreadsheet program, etc. While individually this group of uses might be considered to be exceptions or oddities not worth mentioning, when taken as a group, they form a significant body of applications for this powerful method of storing information.

Unfortunately, many of this last group of uses produce matrices which are neither sparse nor very densely populated, but at the same time do not seem to fall into any of the special cases computer scientists are used to working with. Additionally, even when using matrices for a more traditional problem, one does not always know how big to make the matrix. As a result, one is frequently left with the problem of either imposing unnecessary restrictions on the complexity of problems which can be handled, or alternatively tying up so much memory/swap space as to effectively monopolize the computer (assuming the job will run at all). In both of these cases, there will be regions which are totally devoid of meaningful (e.g., nonzero) data, while other regions have at least some data.

The author refers to this class of matrices as being "semi-sparse." To visualize this concept, consider a map of the Earth. Areas covered by land would be areas with a high data density. Other regions consisting of islands surrounded by water would have a low but nonzero data density. Finally, the middle of large lakes and open stretches of ocean would correspond to a zero data density, and this is what would make the map a candidate for classification as a semi-sparse matrix. Figure 1 should clarify this concept further.

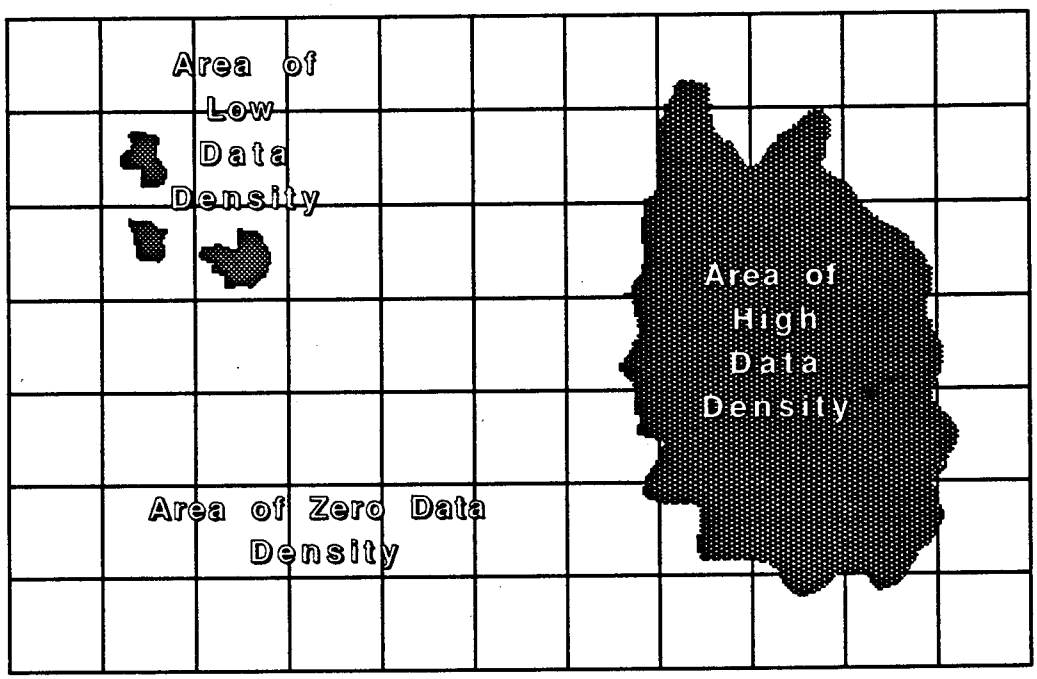

Figure 1. An example of a semi-sparse matrix.

4. DYNAMIC MATRICES: THE THEORY

What is needed is a matrix representation scheme which superimposes a grid on the original matrix in such a way that a grid cell has a high probability of either containing only zeros, or of having a high density of nonzero elements, but a very low probability of having a low (but nonzero) density of nonzero elements. There are several ways in which this problem could be approached. For example, the entire matrix could be made into one grid cell, but this would not be very helpful, since it would put us back where we started from. Alternatively, each element of the matrix could be put in a separate grid cell. While this would clearly meet the goal for the gridding process, the problem of how to store the grid would be equivalent in difficulty to the original problem. Clearly, this is not a useful solution. Another approach would be to use an adaptive grid with cells of varying size, and possibly even geometry so that large cells would be used in large regions of either high or zero density, while smaller cells could be used in regions of low density and in transition zones. Unfortunately, in many cases there is no prior knowledge of what this density will be, and therefore one would not know how to construct such an adaptive grid.

While there may be other successful approaches to the storage problem, the one that the author settled on is to divide the matrix up into a large number of equal-sized rectangular partitions (although the grid cells along some or all of the edges of the matrix may actually extend past the boundary of the matrix). By adjusting the size and shape of these partitions (or as the author refers to them, submatrices), they can be tailored to optimize the performance over the range of cases normally found in a particular application. One can then construct a matrix of pointers with each pointer pointing to a corner of a submatrix, if the submatrix has any nonzero data stored in it, and otherwise pointing to NULL (or some people prefer the word "NIL"). (This, of course, assumes that one always points to the same corner for each submatrix [e.g., the upper left-hand corner], and that the i,j element of the matrix of pointers points to the submatrix storing the elements contained in the i,j cell of the grid.) As more data is added to the matrix (or alternatively, as the matrix is populated with data), attempts to store values in elements belonging to an unpopulated submatrix will result in space being allocated to the submatrix, and initialized to all zeros. After this initialization is completed, the request to store information in an element of that submatrix will be honored. Figure 2 illustrates these ideas.

At this point, there are several issues which should be noted:

- When dealing with semi-sparse matrices, if the size of the submatrices are decreased, the amount of dynamically allocated space needed to handle regions of low, but nonzero, data density will decline.

- The size of the matrix of pointers is inversely proportional to the size of the submatrices. Therefore, for any particular class of problems, there is an optimal range for the size of the submatrices.

- For a matrix stored in either column major or row major format, the address associated with an arbitrary element of this matrix can be calculated using one integer multiplication and one integer addition. For a dynamic matrix, when calculating the address of an arbitrary element of the matrix, one must perform at least two integer divisions, two integer multiplications, and two integer additions. (The current implementation actually performs four integer divisions. One alternative to this would be to do two additional integer multiplications and two additional integer subtractions.) Therefore, it is considerably more time-consuming to access an arbitrary element of a dynamic matrix.

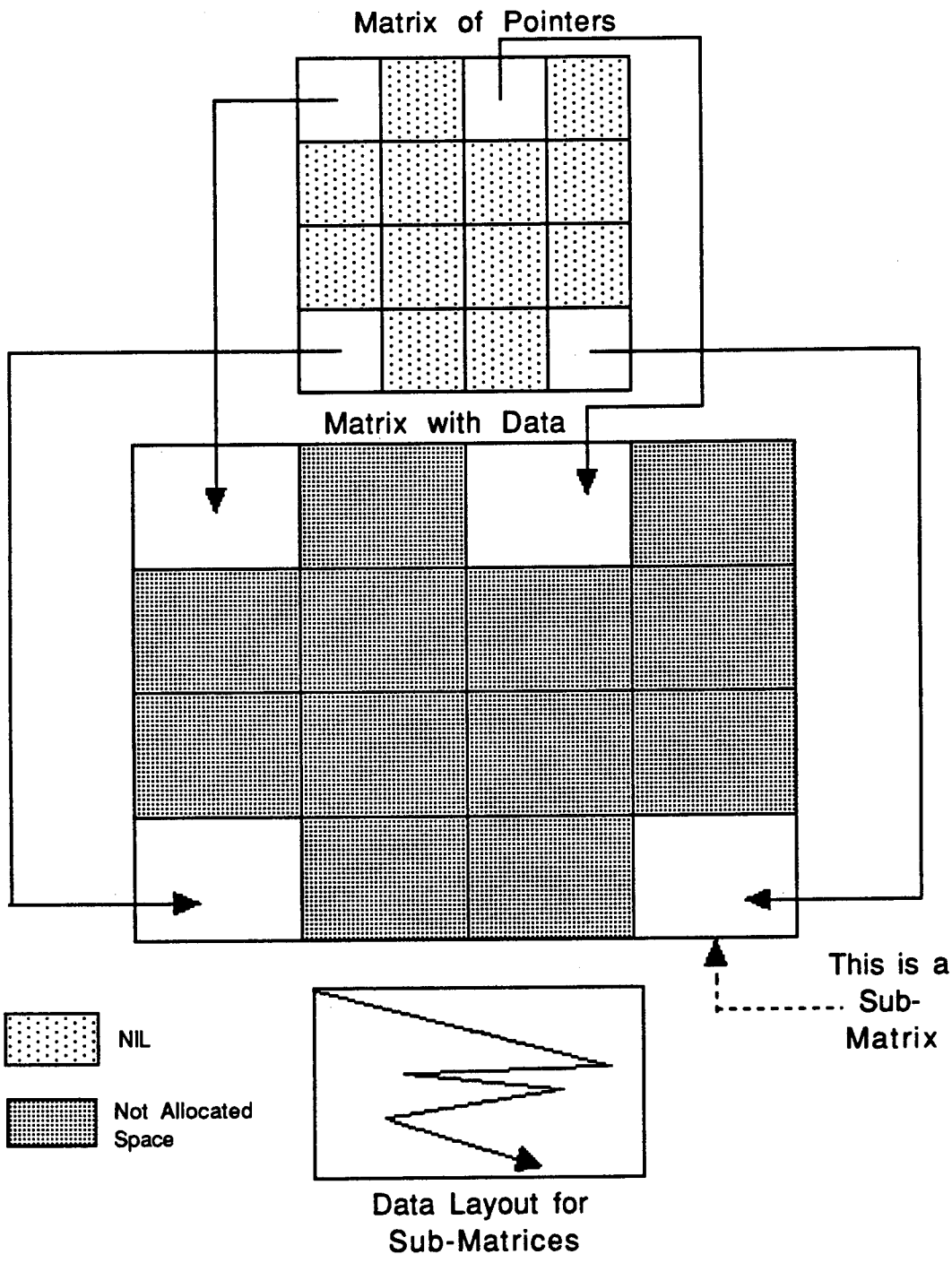

Figure 2. An example of data layout for a dynamic matrix.

- In almost all cases this technique can be used to efficiently represent sparse matrices, dense matrices, or the commonly described special cases, although in many cases, the speed of access will decrease. (Note that in a paged virtual memory environment, this added overhead may be more than offset by a drop in the number of page faults.)

- It is also possible to implement a number of techniques which can substantially decrease the average amount of time it takes to access an element.

As was just noted, in a paged virtual memory environment, the use of dynamic matrices can dramatically reduce the number of page faults. It is, in fact, this particular property which lead the author to develop the concepts discussed in this report. While some computers use a page size of 1-kB or less (e.g., DEC's VAX line of computers use a 512-byte page size), most computers use a somewhat larger page size. (The author has seen many references to systems using page sizes in the range of 2–16 kB, and at least some systems based on the 4.2BSD flavor of UNIX seem to use an open-ended variable page size system where page sizes of 1 MB or more are not uncommon.) If a 1,000 × 1,000 element matrix, with a data element size of 4 bytes, is being manipulated, it should be clear that regardless of whether the matrix is being stored row major or column major, it will take at most 2 pages to store 1,000 elements (assuming a page size of at least 2 kB). In some cases, the system may allocate many thousands of elements to a single page.

There are a number of applications (e.g., matrix multiplication, and drawing a border around a raster image) that will always result in accessing a matrix stored in row major format in column major order or vice versa. The implication of this is that these accesses will constantly cross page boundaries, therefore making a large part of the matrix (if not the entire matrix, depending on the size of the matrix and the size of the pages) part of the working set. Under the appropriate conditions (and with large page sizes these conditions are easily reached), the size of the working set can significantly exceed the size of physical memory.

The concept of a dynamic matrix tries to deal with this problem in three ways. The first is that if one is manipulating sparse or semi-sparse matrices, the amount of space needed to store the matrix may decrease to the point that the working set is smaller than the size of physical memory (this would virtually eliminate the potential for paging in the first place). The second approach is that even if the matrix is densely packed with data, by making the submatrices roughly square (the ratio of length to the width,

probably should be between 0.1 and 10), one eliminates the importance of accessing the matrix in either row major or column major manor. Therefore, under the conditions which would in the past have caused the code to rapidly cross page boundaries (e.g., each access might be to another page in memory, cycling back to the first page only after visiting a thousand or more other pages), will on average cause the code to cross page boundaries much more slowly. The number of successive accesses to the same page of memory might increase by a factor of between 10 and 100, while the number of pages being cycled through might decrease by a similar factor. Finally, if only a small number of rows and columns are needed in memory at any given time for the code to be efficient (e.g., the fifth row of matrix A and the tenth column of matrix B when multiplying the two matrices together), it is likely that the use of dynamic matrices will allow the program to reduce the amount of additional information which must also be stored. Since all of these attributes serve to dramatically reduce the size of the working set, they make dynamic matrices a nearly perfect way to avoid thrashing.

5. DYNAMIC MATRICES: IMPLEMENTATION DETAILS

It is one thing to say that one has developed a new data structure, it is quite another to actually put it to use. Probably the best way to approach this problem would have been to develop a whole new computer language (or a new flavor of an existing language). Unfortunately, that would be a major undertaking and the benefits did not seem to justify the effort. The only other alternative was to use the capabilities of an existing language. Since there are relatively few commonly used general purpose languages which support pointers (in this country, C, PASCAL, and ADA seem to dominate this niche), C seemed to be the best choice for the job (although the generic packages of ADA make that an interesting language to consider).

Once the language is chosen, a method of implementation must be selected. There are, roughly speaking, three ways to proceed:

- The code could be directly inserted in line without making any attempt to have the code maintain a separate identity. While this might produce slightly faster programs, considering the complexity of this technique, it would make it much more difficult to maintain the code, or to use this technique in other programs.

- Alternatively, everything could be implemented using functions, subroutines, procedures, or whatever one is used to calling them. This approach would provide the greatest level of data hiding, and therefore, would probably be the safest. Unfortunately, on many systems, the overhead associated with function calls would make this fairly slow.

- The alternative which was finally selected was to construct a set of macros using C's macro capabilities. This introduced a number of problems inherent with the use of macros, but in the end produced fairly efficient code that, with a modest amount of caution (e.g., macro arguments must not have any side effects associated with them), could be safely used. The entirety of this set of macros is included in Appendix A.

There are four basic functions which should be included in this group of macros. These functions are to:

- allocate/initialize a dynamic matrix;

- deallocate a dynamic matrix (although the author has rarely used this macro);

- retrieve a value from an element of the matrix; and

- store a value in an element of the matrix.

The allocation macro dynamically allocates a matrix of pointers (using malloc and calloc) based on information concerning the size and shape of the matrix and the size and shape of the submatrices (the original version assumes all elements are integers, although it is simple enough to produce versions of these macros for other data types). It also records the default value for the elements of the matrix (normally zero, but in some special cases other values may be more appropriate).

The deallocation macro deallocates the space reserved for each of the submatrices and then for the matrix of pointers.

Fetching (retrieving) a value from the matrix involves calculating the address of the pointer to the submatrix. If the submatrix exists, then the address of the relevant element must be calculated and the value of that element returned. If the submatrix does not exist, then the default value is returned. This is a rather straightforward process; the catch is that all of these actions would normally involve multiple statements, making it impossible to use the returned value in an equation. By using some of the unique capabilities of C, this problem was overcome, although it should be noted that several of the macros in this group use globally defined variables. Additionally, one should be careful when using this macro and store the results of any calculations involving the value it returns in a normal variable before continuing on. The value stored in the normal variable can then be used to update the dynamic matrix should the need arise.

The macro for storing a value in the dynamic matrix is probably the most complex, since it requires several statements to implement (this also means that it cannot appear to the left of an equal sign). It starts out like the fetch routine, by calculating the address of the pointer to the submatrix. If the value of the pointer is non-null, the address of the specified element is calculated and the value is stored at that location. Otherwise, it is necessary to allocate space for a new submatrix, initialize the submatrix, and then store a value in that submatrix. Since this can get a bit complicated, the interested reader should review the code in Appendix A for the macros and Appendix B for examples of how to use these macros.

At this point, a functional package exists for creating and manipulating a dynamic matrix. Unfortunately, experience has shown that if one stops at this point, the overhead may be considered to be a bit excessive. Therefore, the next section will discuss some additional ways that can be used to improve the efficiency of these matrices.

6. DYNAMIC MATRICES: PERFORMANCE ENHANCEMENTS

While there are several approaches to enhancing the performance of these matrices, the efforts generally fall into two categories—reduce the amount of time spent calculating addresses, or reduce the number of times the matrix is accessed in the first place. Theoretically, there are a number of ways one could try to remember the address of a submatrix from one macro call to the next, and then check to see if the same submatrix is being accessed. Unfortunately, most of these techniques are either special purpose (e.g., primarily relevant to matrix multiplication, but not to rasterization or vice versa), or have a sufficiently high cost associated with them as to make them of questionable value. One exception to

this rule is to implement something akin to the C constructs A + = 2, A * = 2, etc. This is a relatively general purpose construct, which if fully implemented will eliminate half of the address calculations with no offsetting computational burden (although some people will complain about what this does to the clarity of this code). This idea was implemented as the UPDATE macro, and since it is a macro and not a function/subroutine, it was very easy to allow the compiler to deal with the problem of which operator was specified.

Another general method for speeding up the code is to restrict the size of the submatrices to a fixed size and to make the dimensions of the submatrices be powers of two. In this way, the integer multiplication and division can be eliminated by the use of bit shifting, and the modulus operator can be replaced with bit masks. All of this can be done on a machine-independent basis and may improve the speed of the code by as much as a factor of 10 or more (this claim is based on the fact that many machines implement integer division and the modulus operator using very slow functions, and that even when there is direct hardware support for these operations, they may still be some of the slowest operations implemented in hardware).

The alternative approach of reducing the number of times the matrix is accessed appears to be much more successful. Remembering that dynamic matrices were originally intended for use with semi-sparse matrices, it should be clear that if one can check to see if a submatrix has been allocated, that assuming it has not been allocated, one can save a significant amount of time. One way that this might be used is in printing out the matrix. One would simply print out the default value for the next N values, where N is the number of columns in the submatrix. Similarly, when performing matrix multiplication, this could allow one to eliminate a great number of calculations. This concept is the basis for the INQUIRE macro.

Along the same line of reasoning, if one knows that a submatrix exists, and that all of it's elements will be accessed in the next set of operations (e.g., matrix multiplication might be a prime example of this), one can fetch the starting address of the submatrix and directly manipulate its elements. Clearly, this is not something that the casual programmer is likely to want to do, but for someone familiar with writing sophisticated programs, this should not be such a daunting process. This has the advantage of simplifying the effort needed to access an element, back to roughly that associated with regular matrices, and can, therefore, produce a major cost savings. This construct has been implemented as the FETCH_POINTER macro.

There are a number of other possible ways to reduce the number of times a dynamic matrix is directly accessed (e.g., something similar to the PACK and UNPACK procedures in PASCAL). At the present time, no attempt has been made to investigate any other possibilities. When compared to a program written using regular matrices (assuming that the matrices are relatively small), a program using dynamic matrices to store a semi-sparse matrix is likely to run six to ten times slower. However, in the case of rasterizing wireframe data, it appears to rarely use more than one-fourth the amount of memory. Therefore, if the program written using regular matrices has a working set that is no larger than four times the size of main memory, then the program written using dynamic matrices may be 10,000 or more times faster. Actual numbers are hard to come by, since in the case which led to the development of these theories, the traditional program repeatedly tied up the system it was running on for over an hour before the computer finally crashed. On the other hand, a similar program (the author did not have access to the source code of the program which crashed the computer) implemented with dynamic matrices "usually" finishes the job in about 10–15 minutes on the same computer. The reason for saying "usually" is that the performance depends on the data contained in the matrix, and if the matrix is not semi-sparse, or if a poor size and shape were selected for the submatrices, the run might take significantly longer (possibly even crashing the system).

7. CONTINUING RESEARCH

There are ongoing efforts to use dynamic matrices to multiply large, square, dense matrices on a variety of machines. While the results are still preliminary, they appear to be quite promising. In particular, by choosing a relatively small size for the submatrices, one can dramatically improve the hit rate for the high-speed cache memory most systems are now equipped with. This may also make it easier to optimize the code for other aspects of a particular machine's architecture. While the use of smaller submatrices has the potential for increasing the amount of paging, experience has indicated that for most machines, the computational burden is likely to become excessive long before this becomes a problem. Appendix C contains a copy of the macros used in this effort.

Another area which appears to be interesting is to port dense matrix versions of this code to Fortran 90. Since Fortran 90 does have a limited amount of support for dynamic memory allocation, it should be possible to implement this efficiently. Unfortunately, it is not clear if this support is sufficient to allow the author to implement support for sparse and semi-sparse matrices. Even so, the reduction in the size of the working set and the improvement in cache utilization may justify the effort.

8. FUTURE DIRECTIONS: THE EXTENSIBLE DYNAMIC MATRIX

One potentially interesting extension of these ideas is to merge the traditional linked list representation of sparse matrices with the author's concept of the dynamic matrix. In this way, one might be able to allow the matrix to accommodate a wider range of data sets without the need to allocate a large matrix of pointers. This might be accomplished by using linked lists to implement a multidimensional mesh of matrices of pointers. Initially, only one element in this mesh would be allocated. As the size of the matrix grows, additional matrices of pointers can be allocated as needed. This would allow the same data structure to efficiently handle storage requirements which might vary by several orders of magnitude from one run to the next. The author refers to this idea as an "extensible dynamic matrix."

Earlier, this report alluded to the ability to link together more than one matrix of pointers. There are several strategies which could be used at this point for determining the size of additional matrices of pointers. Probably the simplest strategy would be to use a constant size for the matrices of pointers. While this would be the easiest to implement, it might not be efficient if the size of the matrix is to grow by more than a factor of 10 in any direction. A slightly more complicated alternative is to use one or more fixed sizes for the additional matrices of pointers, with the size getting larger as one adds on more matrices of pointers. This would help to limit the maximum length of the list of linked lists, but might require more programmer involvement than is desirable. Probably the two most desirable alternatives are to either have the size of the matrices of pointers and/or the size of the submatrices grow as a power of two as one moves further away from the original matrix of pointers. In this way, one is guaranteed that the size of the linked list will always remain small.

At this point, a purist is likely to point out that this will result in a return to O(N) behavior for the time required to access any given element. The author believes that most readers will agree that this dependency is so small that effectively it should be possible to treat the extensible dynamic matrix as though it has O(1) access time. An example of this is that if one starts out with a 16×16 element matrix of pointers with an 8×8 element submatrix, one will already be able to store 16 k elements in the matrix. If one adds on the eight adjacent matrices of pointers, and uses the strategy for doubling the dimension each time one moves away from the original matrix of pointers, then one can now store nearly 150 k elements. Taking this out just one more level, one is nearly up to 3 M elements. Finally, using linked lists with a maximum length of 6 nodes (the length of the linked lists for elements at the corners might actually be 12 nodes, depending on how things are implemented), one can store 1 G elements.

Additionally, if this were a three-dimensional matrix, the numbers would be growing even faster. Since there are very few computers currently in production which could efficiently either store or manipulate matrices of these sizes (unless the matrix is either highly sparse or decidedly nonsquare), this should probably be considered to be the worst-case scenario for the length of the linked lists. Clearly, if this is the worst case, then for most practical purposes, this is not a problem, and one returns to the author's earlier conclusion that effectively the extensible dynamic matrix has an O(1) access time.

9. CONCLUSIONS

With several years worth of experience applying dynamic matrices to the problem of manipulating and creating large raster images (and very encouraging early results from the application of dynamic matrices to the multiplication of large square matrices), the author feels quite confident in calling dynamic matrices a proven success. Hopefully, this report will interest others to pursue additional avenues of investigation using these techniques. Additionally, the author believes that the initial results which indicate that the use of dynamic matrices can result in substantially higher cache hit rates, with the associated increase in perceived processor performance, should encourage compiler writers to investigate more sophisticated methods for storing matrices. To some extent, this is already being done on scalably parallel processors where the compilers support layout or tiling directives. However, the author believes that this work indicates that high-end workstations with only a few processors (possibly only one) can also benefit from these ideas. In fact, the original development work in this area was done on 2 MIP, 16-MB SUN 3 uniprocessors.

INTENTIONALLY LEFT BLANK.

APPENDIX A: THE DYNAMIC MATRIX MACROS FOR INTEGER DATA

The following group of macros are normally found in a file called dyn.h and represent the original group of macros used in the development of the author's rasterization software. As presently configured, they are designed to store integer data, and it is assumed that an integer is 32 bits. The 32 bit assumption comes into play in two ways. The first way is unique to the rasterization software in that all of its bit manipulations are based on the assumption of 32 bit integers. The second way may be of more general interest in that if significantly fewer than 32 bits are used to represent integers (e.g., 16 bits), some of the address calculations may not work properly. It is also theoretically possible that there could be problems on machines with 64 bit addresses if 32 bit integer arithmetic is used to calculate offsets for very large matrices (e.g., well in excess of 100 million elements). Fortunately there are very few machines at the present time which could realistically manipulate matrices of this size, and some of them have already solved that problem by using 64 bit integers.

```
/*************************************************************/
/*                                                           */
/*  This include file was written by Daniel M. Pressel at    */
/*  CRDEC in August of 1988.  It is intended to provide a    */
/*  uniform set of structure definitions and macros for      */
/*  implementing in C his concept of a dynamic matrix.       */
/*                                                           */
/*************************************************************/
/*                                                           */
/*  In this file, the following items are defined:           */
/*                                                           */
/*       1)  A scratch variable.  For uniqueness, it is      */
/*           called DYNAMIC_TEMP.                            */
/*                                                           */
/*       2)  Two named constants, USED and UNUSED, which     */
/*           will be used as boolean values in some of       */
/*           the macros to follow.                           */
/*                                                           */
/*       3)  A type definition for the structure used to     */
/*           simulate my concept of dynamic matrices.        */
/*                                                           */
/*       4)  The following macro definitions:                */
/*                                                           */
/*             a)  ALLOCATE:  This macro consists of a       */
/*                 series of seperate statements used to     */
/*                 allocate and initialize the core          */
/*                 structure for a dynamic matrix.           */
/*                                                           */
/*             b)  FETCH_VALUE:  Unlike  ALLOCATE, this      */
/*                 macro is composed of a single             */
/*                 arithmetic expression.  As such, it is    */
/*                 suitable for inclusion directly into      */
/*                 an arithmetic expression.  To achieve     */
/*                 this goal, both the comma operator and    */
/*                 the ternary conditional operator were     */
/*                 used.  As such this is an extremely       */
/*                 complicated macro.                        */
/*                                                           */
/*
```

```
                c)  FETCH_POINTER_TO_BLOCK:  This macro is */
/*                  also designed to be directly included  */
/*                  into arithimetic expressions and       */
/*                  function calls.  Given a row and       */
/*                  column number, this macro will         */
/*                  determine in which of the blocks the   */
/*                  matrix was subdivided into, the        */
/*                  element resides, it then returns a     */
/*                  pointer to the block of memory         */
/*                  containing this element.  NOTE:  If    */
/*                  the specified block has not had any    */
/*                  values stored into it, then the NULL   */
/*                  pointer is returned.  It is therefore  */
/*                  imperative that the                    */
/*                  INQUIRE_POINTER_TO_BLOCK macro be used */
/*                  first to see if the block exists.      */
/*                                                         */
/*              d)  INQUIRE_POINTER_TO_BLOCK:  This macro  */
/*                  returns a boolean value of either USED */
/*                  or UNUSED, depending upon if anything  */
/*                  other than the default value is stored */
/*                  in the elements of the specified       */
/*                  block.  It may be used in an           */
/*                  expression, but probably is best used  */
/*                  as part of if, while, and for          */
/*                  statements.                            */
/*                                                         */
/*              e)  STORE_VALUE:  Here is another macro,   */
/*                  which is composed of multiple          */
/*                  statements, and must not be included   */
/*                  in any expressions.  It will update    */
/*                  the value of an element in a dynamic   */
/*                  matrix, allocating space for a block   */
/*                  as needed.  When a block is allocated, */
/*                  this macro will also initialize each   */
/*                  of its elements.  Since this takes     */
/*                  some time, and increases the amount of */
/*                  memory in use,  this macro will ignore */
/*                  requests to store the default value    */
/*                  into an element of the matrix if it    */
/*                  resides in a previously unused block.  */
/*                                                         */
/*              f)  DEALLOCATE:  This is a multi statement */
/*                  macro used to free all of the memory   */
/*                  previously used by a dynamic matrix.   */
/*                                                         */
/*              g)  UPDATE_VALUE:  This combines the       */
/*                  functionallity of FETCH_VALUE with     */
/*                  STORE_VALUE.  There are two added      */
/*                  arguments.  The second extra argument  */
/*                  contains a binary operator which will  */
/*                  be applied to value stored in the      */
/*                  dyanmic matrix.  The first additional  */
/*                  argument contains a value or           */
/*                  expression which will come to the      */
/*                  right of the binary operator.  This    */
```

```
/*                  macro is not required, but its use     */
/*                  may result in faster running code.     */
/*                                                         */
/*  NOTE:  As with all macros, one should never allow any  */
/*  of the arguments to the macros to have side effects.   */
/*  If side effects are present, they will almost always   */
/*  take effect several times, with usually undesirable    */
/*  consequences.  Additionally, one may include           */
/*  expressions as part of most of the arguments, but      */
/*  care must be taken in doing so.  If the expressions    */
/*  are computationally intensive, or call functions they  */
/*  may have severe negative speed consequences.  If this  */
/*  becomes a problem, the use of temporary or scratch     */
/*  variables would be well advised.                       */
/*                                                         */
/***********************************************************/

/***********************************************************/
/*                                                         */
/* This software is the property of the United States      */
/* Government.  When and if it is made available for use   */
/* by others, it is totally at their own risk.             */
/* Furthermore, the author is unable to guarantee          */
/* continued support for this product.  Any modifications  */
/* are the responsibility of the users.  Finally, it is    */
/* illegal to sell either this software, or any            */
/* derivative software, to other United States Government  */
/* installations without first informing them that the     */
/* software is already the property of the United States   */
/* Government.                                             */
/*                                                         */
/***********************************************************/


char *malloc();
char *calloc();
int  *DYNAMIC_TEMP;
int   DYNAMIC_TEMP_VALUE;
#define USED 1
#define UNUSED 0

typedef struct
   {
    int initial_value;
    int number_of_rows;
    int number_of_columns;
    int **pointer_to_matrix;
    int rows_in_block;
    int columns_in_block;
    int blocks_in_row;
   }   DYNAMIC_MATRIX;

#define ALLOCATE(pointer,init_val,num_of_rows,\
           num_of_columns,num_rows_in_block,\
           num_columns_in_block) \
         pointer=(DYNAMIC_MATRIX *)malloc(sizeof\
```

```
            (DYNAMIC_MATRIX));\
         pointer -> initial_value=init_val;\
         pointer -> number_of_rows=num_of_rows;\
         pointer -> number_of_columns=num_of_columns;\
         pointer -> blocks_in_row=(num_of_columns +\
            num_columns_in_block -1 )/num_columns_in_block;\
         pointer -> pointer_to_matrix=(int **)\
            calloc(((num_of_rows + num_rows_in_block - 1)\
            /num_rows_in_block)*(pointer ->  \
            blocks_in_row),sizeof(int *));\
         pointer -> rows_in_block=num_rows_in_block;\
         pointer -> columns_in_block=num_columns_in_block;

#define FETCH_VALUE(pointer,row_number,column_number) \
         ((DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
            (row_number) / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)/\
            (pointer -> columns_in_block))),\
            ((DYNAMIC_TEMP != 0) ? *((DYNAMIC_TEMP) +\
            (row_number) % (pointer -> rows_in_block) *\
            (pointer -> columns_in_block ) +\
            (column_number) % (pointer -> columns_in_block))\
            : (pointer -> initial_value)))

#define FETCH_POINTER_TO_BLOCK(pointer,row_number,\
           column_number) \
         (*(pointer -> pointer_to_matrix + (row_number) \
            / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)/\
            (pointer -> columns_in_block)))


#define INQUIRE_POINTER_TO_BLOCK(pointer,row_number,\
            column_number) \
         ((*(pointer -> pointer_to_matrix + (row_number) \
            / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)/\
            (pointer -> columns_in_block)))\
            != NULL ? USED : UNUSED )

#define STORE_VALUE(pointer,row_number,column_number,\
            value) \
         DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
            (row_number) / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)\
            / (pointer -> columns_in_block)); \
         if (DYNAMIC_TEMP != 0) \
         *((DYNAMIC_TEMP) + (row_number) %\
         (pointer -> rows_in_block) *\
         (pointer -> columns_in_block ) + (column_number)\
         % (pointer -> columns_in_block)) = (value) ; \
      \
      else \
         if ( (value) !=  (pointer -> initial_value))\
            {\
            register int DYNAMIC_COLUMN,\
```

```
                    DYNAMIC_INITIAL=\
                       (pointer -> initial_value),\
                       DYNAMIC_ROW;\
                 if (DYNAMIC_INITIAL == 0)\
                    DYNAMIC_TEMP=(int *)\
                       calloc((pointer -> rows_in_block) * \
                       (pointer -> columns_in_block),\
                       sizeof(int));\
                 else\
                    {\
                     DYNAMIC_TEMP=(int *)\
                        malloc((pointer -> rows_in_block) * \
                        (pointer -> columns_in_block) *\
                        sizeof(int));\
                     *(pointer -> pointer_to_matrix +\
                        (row_number)/\
                        (pointer -> rows_in_block) *\
                        (pointer -> blocks_in_row) \
                        + (column_number)/\
                        (pointer -> columns_in_block)) \
                        =DYNAMIC_TEMP; \
                     \
                    for(DYNAMIC_ROW=0; DYNAMIC_ROW <\
                       (pointer -> rows_in_block) ;\
                       DYNAMIC_ROW++)\
                       for(DYNAMIC_COLUMN=0; DYNAMIC_COLUMN <\
                          (pointer -> columns_in_block) ;\
                          DYNAMIC_COLUMN++) \
                          \
                          *(DYNAMIC_TEMP + DYNAMIC_ROW * \
                             (pointer -> columns_in_block) \
                             + DYNAMIC_COLUMN) =\
                             DYNAMIC_INITIAL;\
                    };\
                 *(pointer -> pointer_to_matrix +\
                    (row_number) / (pointer -> rows_in_block)\
                    * (pointer -> blocks_in_row) \
                    + (column_number)/\
                    (pointer -> columns_in_block)) \
                    = DYNAMIC_TEMP; \
                 *((DYNAMIC_TEMP) + (row_number) %\
                    (pointer -> rows_in_block)\
                    * (pointer -> columns_in_block ) +\
                    (column_number) % \
                    (pointer -> columns_in_block)) = (value);\
              };;

#define DEALLOCATE(pointer) \
           { \
            register int column,row; \
            for ( row=0 ; row < (pointer -> number_of_rows\
               + pointer -> rows_in_block - 2); \
               row+=(pointer -> rows_in_block)) \
               \
               for ( column=0 ; column <\
                  (pointer -> number_of_columns + \
                  pointer -> columns_in_block - 2); \
```

```
                column+=(pointer -> columns_in_block)) \
                \
                free(*((pointer ->\
                   pointer_to_matrix)++)); \
          };
#define UPDATE_VALUE(pointer,row_number,column_number,\
          value,operation) \
        DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
          (row_number) / (pointer -> rows_in_block) *\
          (pointer -> blocks_in_row) + (column_number)\
          / (pointer -> columns_in_block)); \
        if (DYNAMIC_TEMP != 0) \
            DYNAMIC_TEMP_VALUE = (*((DYNAMIC_TEMP) +\
              (row_number) % (pointer -> rows_in_block)\
              * (pointer -> columns_in_block ) \
              + (column_number) %\
              (pointer -> columns_in_block))) \
              operation (value) ; \
        \
        else \
            DYNAMIC_TEMP_VALUE =\
              (pointer -> initial_value) operation\
              (value);\
        \
        if (DYNAMIC_TEMP != 0 ) \
          *((DYNAMIC_TEMP) + (row_number) %\
            (pointer -> rows_in_block) * \
            (pointer -> columns_in_block ) +\
            (column_number) % \
            (pointer -> columns_in_block)) =\
            DYNAMIC_TEMP_VALUE ; \
        \
        else \
          if ( DYNAMIC_TEMP_VALUE !=\
            (pointer -> initial_value))\
            {\
             register int DYNAMIC_COLUMN,\
               DYNAMIC_INITIAL=\
                 (pointer -> initial_value),\
                 DYNAMIC_ROW;\
             if (DYNAMIC_INITIAL == 0)\
               DYNAMIC_TEMP=(int *)\
                 calloc((pointer -> rows_in_block)\
                 * (pointer -> columns_in_block)\
                 ,sizeof(int));\
             else\
               {\
                DYNAMIC_TEMP=(int *)\
                  malloc((pointer -> rows_in_block)\
                  * (pointer -> columns_in_block)\
                  * sizeof(int));\
                *(pointer -> pointer_to_matrix +\
                  (row_number)/ \
                  (pointer -> rows_in_block) *\
                  (pointer -> blocks_in_row) \
                  + (column_number)/\
```

```
            (pointer -> columns_in_block)) \
            =DYNAMIC_TEMP; \
          \
        for(DYNAMIC_ROW=0; DYNAMIC_ROW <\
           (pointer -> rows_in_block) ;\
           DYNAMIC_ROW++)\
           for(DYNAMIC_COLUMN=0;\
              DYNAMIC_COLUMN < \
              (pointer -> columns_in_block) ;\
              DYNAMIC_COLUMN++) \
            \
            *(DYNAMIC_TEMP + DYNAMIC_ROW * \
               (pointer -> columns_in_block) \
               + DYNAMIC_COLUMN) =\
               DYNAMIC_INITIAL;\
    };\
 *(pointer -> pointer_to_matrix +\
    (row_number)/(pointer -> rows_in_block)\
    * (pointer -> blocks_in_row) \
    + (column_number)/\
    (pointer -> columns_in_block)) \
    = DYNAMIC_TEMP; \
 *((DYNAMIC_TEMP) + (row_number) %\
    (pointer -> rows_in_block)\
    * (pointer -> columns_in_block ) +\
    (column_number) % \
    (pointer -> columns_in_block)) =\
    DYNAMIC_TEMP_VALUE ; \
};;
```

INTENTIONALLY LEFT BLANK.

APPENDIX B: RASTERIZATION SOFTWARE USING DYNAMIC MATRICES

The following are three of the routines used to turn Calcomp plot files (using a subset of the Calcomp 960 plotter command set) into raster format suitable for use with a 400 dot per inch Versatec electrostatic plotter. The routines listed here, use, in one way or another, macros from the Dynamic Matrix package listed in Appendix A to create and plot raster images for the previously mentioned plotter. The missing routines do not make use of these macros and have been left out for the sake of brevity. This is only one of several similar programs created using these macros. In all, programs to drive HP laserjets, Seiko color hard copy units, 200 dot per inch monochrome Versatec electrostatic plotters, and 400 dot per inch color and monochrome Versatec electrostatic plotters were written. Over a six-year period, these programs became so popular at what was then CRDEC that over 10,000 monochrome 400 dot per inch plots (roughly half of these were either D or E size) were made. Additionally, smaller numbers of color and 200 dot per inch plots were also made. The number of plots to HP laserjets is impossible to estimate, but this appears to be the most popular of all of the supported plotters.

```
/*************************************************************/
/*                                                           */
/*  This program was written by Daniel M. Pressel at         */
/*  CRDEC in the summer of 1989.  It is designed to take     */
/*  a plot file in Calcomp 960 format with software text     */
/*  and rasterize it for a 400 dpi monochrome 36 inch        */
/*  Versatec electrostatic plotter.  In writing this         */
/*  program, Mr. Pressel has made extensive use of his       */
/*  concept of a dynamic matrix.                             */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*                                                           */
/*  This software is the property of the United States       */
/*  Government.  When and if it is made available for use    */
/*  by others, it is totally at their own risk.              */
/*  Furthermore, the author is unable to guarantee           */
/*  continued support for this product.  In fact this        */
/*  package has only been installed on systems running       */
/*  rev 3.0.  The installation instructions may only         */
/*  apply to this rev and probably need modifications if     */
/*  the software is to be successfully installed on          */
/*  future revs.  Any such modifications are the             */
/*  responsibility of the administrators at the sites        */
/*  receiving copies of this software.  Finally, it is       */
/*  illegal to sell either this software, or any             */
/*  derivative software, to other United States              */
/*  Government installations without first informing         */
/*  them that the software is already the property of the    */
/*  United States Government.                                */
/*                                                           */
/*************************************************************/


#include <stdio.h>
#include <string.h>
#include "dyn.h"
#include "calcomp_defines.h"
```

```
main()
{
 DYNAMIC_MATRIX *image;

 char type;
 char firstb;
 char secondb;

 int calcomp_get_command();

 int delta_x,delta_y;
 int old_x,old_y;
 int max_x,max_y;
 int bit;
 int row,column;

#if RESOLUTION == 200
 int offset=0;
#else
 int offset=1;
#endif

 int step;

#if RESOLUTION == 200
 int thickness=1;
#else
 int thickness=3;
#endif

 int pen_status=UP;
 int number_of_points;
 register int loop_counter;

/*************************************************************/
/*                                                           */
/* The array hpstylen stores the pattern for "horizontal     */
/* lines" (abs(slope) < 1).                                  */
/*                                                           */
/*************************************************************/

 unsigned int hpstyle;

/*************************************************************/
/*                                                           */
/* The array vpstyle stores the pattern for "vertical        */
/* lines" abs(slope) >= 1).                                  */
/*                                                           */
/*************************************************************/

 unsigned int vpstyle[33];

 float scale;

 void calcomp_raster();
 void calcomp_plot();
```

```
/*********************************************************/
/*                                                       */
/*  Do all kinds of initializations prior to the start of */
/*  the run.                                             */
/*                                                       */
/*********************************************************/

#if RESOLUTION == 200
 hpstyle=0x80000000;
 vpstyle[1]=0x80000000;
#else
 hpstyle=0xe0000000;
 vpstyle[1]=0x80000000;
 vpstyle[2]=0x80000000;
 vpstyle[3]=0x80000000;
#endif

 scale=RESOLUTION * .0004921;
 step= 1/scale;
 old_x=0;
 old_y=0;
 max_x=old_x;
 max_y=old_y;

 ALLOCATE(image,0,MAX_ROWS,MAX_COLUMNS,ROWS_IN_BLOCK,
    COLUMNS_IN_BLOCK);

     while ( calcomp_get_command(&number_of_points,
        &pen_status,&type)
        != DONE )
        switch (type)
           {

        /*     case 0x7F:
                goto FAST_OUT;*/
             case 0x20:
             case 0x40:

 /*********************************************************/
 /*                                                       */
 /* Process polyline commands.                            */
 /*                                                       */
 /*********************************************************/

                for (loop_counter=0; loop_counter <
                   number_of_points; loop_counter++)

                   {

 /*********************************************************/
 /*                                                       */
 /* Draw from the current location to the location        */
 /* specified.  This command always uses relative         */
 /* positioning.  It is therefore neccessary to look for  */
 /* and handle negative values.  Otherwise, the           */
 /* coordinates are similar to the move command above.    */
```

```
/*                                                              */
/**************************************************************/

                switch (type)
                   {

                    case 0x20:
                       delta_x=getchar();
                       delta_y=getchar();
                       if ( delta_x > 0x7f )
                          delta_x=(( ~delta_x & 0xff) +
                             1 ) * -1;
                       if ( delta_y > 0x7f )
                          delta_y=(( ~delta_y & 0xff) +
                             1 ) * -1;
                    break;

                    case 0x40:
                       firstb=getchar();
                       secondb=getchar();
                       delta_x=((firstb & 0xff) << 8) |
                          (secondb & 0xff);
                       firstb=getchar();
                       secondb=getchar();
                       delta_y=((firstb & 0xff) << 8) |
                          (secondb & 0xff);
                       if ( delta_x > 0x7fff )
                          delta_x=(( ~delta_x & 0xffff)
                             + 1 ) * -1;
                       if ( delta_y > 0x7fff )
                          delta_y=(( ~delta_y & 0xffff)
                             + 1 ) * -1;
                    break;
                   };

                if ((((old_x+delta_x)*scale+WORD_SIZE) >
                   MAX_ROWS) || (((old_y+delta_y)*scale+
                   WORD_SIZE) > MAX_COLUMNS*WORD_SIZE) ||
                   (((old_x+delta_x)*scale+WORD_SIZE) <
                   0) || (((old_y+delta_y)*scale+
                   WORD_SIZE) < 0))
                   {
                    fprintf(stderr,
                       "Your plot is too big!!\n");
                    fprintf(stderr,
                       "(%d,%d)  max(%d,%d)\n",
                       old_x+delta_x,
                    old_y+delta_y,max_x,max_y);
                    goto FAST_OUT;
                   };

                if (pen_status == DOWN)
                   calcomp_raster(delta_x,delta_y,
                      hpstyle,image,offset,old_x,old_y,
                      scale,step,thickness,vpstyle);
                old_x=old_x+delta_x;
                old_y=old_y+delta_y;
```

```
#ifdef DIAG
                        row=old_x*scale+WORD_SIZE;
                        column=old_y*scale-offset+WORD_SIZE;
                        bit=column%WORD_SIZE;
                        column=column/WORD_SIZE;
                        UPDATE_VALUE(image,row,column,
                           0xFFFF0000 >> bit , | );
                        column++;
                        UPDATE_VALUE(image,row,column,
                           0xFFFF0000 << (WORD_SIZE - bit ),
                           | );
#endif

                      if ( max_x < old_x )
                         max_x=old_x;
                      if ( max_y < old_y )
                         max_y=old_y;

                      };
               };

 FAST_OUT:  ;

/*************************************************************/
/*                                                           */
/*  We have now processed all of the input data.  After      */
/*  calculating the row and column coordinates for each      */
/*  max_x,max_y (with a bit of padding to make it look       */
/*  nicer).  We will then call the plot routine.             */
/*                                                           */
/*************************************************************/


      max_x=max_x*scale + RESOLUTION * 2;
      if ( max_x > MAX_ROWS - 1 )
         max_x=MAX_ROWS - 1;
      max_y=(((max_y * scale) / WORD_SIZE ) + 2 ) +
         COLUMNS_IN_BLOCK - 1;
      if ( max_y >  MAX_COLUMNS )
         max_y = MAX_COLUMNS;
      calcomp_plot(image,max_x,max_y);
}
```

The following routine sends the finished raster image to the plotter. Since the raster image is no longer stored as one long series of bytes, it is impossible to output a single large stream of bytes. Instead, the portion of each sub-matrix which belongs to the current scan line must be output with zero fill for those portions of a scan line which were never allocated space. To the extent that it seemed reasonable to do, this process has been optimized to reduce the number of separate calls to the I/O routines/macros. This also had the effect of reducing the overhead associated with the use of Dynamic Matrices. Several service technicians have reported that it also seemed to produce a higher quality of output than that obtained on similar equipment at other sites. They seemed to feel that this had to do with this software's ability to generate output at a relatively high (roughly 200,000 bytes per second on a 2 MIP SUN 3 with 16 Mbytes of memory) and uniform rate of speed. Presumably this is the result of avoiding disk I/O (either implicit in the form of paging or explicit in the form of reading the raster image in from a file) while at the same time having a relatively low overhead associated with reconstructing the image (compared to using more traditional data compression algorithms).

```
/*************************************************************/
/*                                                           */
/*  This routine was written by Daniel M. Pressel in         */
/*  October of 1988 at CRDEC.  It is part of his versatec    */
/*  plotter package.  It is designed to output the raster    */
/*  image stored in one of his "DYNAMIC MATRICES" to a       */
/*  versatec plotter.                                        */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*                                                           */
/*  This software is the property of the United States       */
/*  Government.  When and if it is made available for use    */
/*  by others, it is totally at their own risk.              */
/*  Furthermore, the author is unable to guarantee           */
/*  continued support for this product.  In fact this        */
/*  package has only been installed on systems running       */
/*  rev 3.0.  The installation instructions may only         */
/*  apply to this rev, and probably need modifications if    */
/*  the software is to be successfully installed on          */
/*  future revs.  Any such modifications are the             */
/*  responsibility of the administrators at the sites        */
/*  receiving copies of this software.  Finally, it is       */
/*  illegal to sell either this software, or any             */
/*  derivative software, to other United States              */
/*  Government installations without first informing them    */
/*  that the software is already the property of the         */
/*  United States Government.                                */
/*                                                           */
/*************************************************************/


#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/vcmd.h>
#include "dyn.h"
#include "calcomp_defines.h"

void calcomp_plot(image,max_x,max_y)
```

```
DYNAMIC_MATRIX *image;
int max_x,max_y;
{
 register int x,y;

 static int dummy[MAX_COLUMNS];

 int temp;

 char vpbuf[BUFSIZ];
 static int plotmd[]={VPLOT,0,0};
 static int prtmd[]={VPRINT,0,0};

/*************************************************************/
/*                                                           */
/* Force a formfeed to get a clean sheet of paper into       */
/* the plotter/printer and then stick the plotter/printer    */
/* into graphics mode.                                       */
/*                                                           */
/*************************************************************/

 temp=setbuf(stdout,vpbuf);

 temp=fflush(stdout);
 sleep(3);

 temp=ioctl(fileno(stdout),VSETSTATE,prtmd);
 temp=fflush(stdout);
 sleep(3);
 putc(4,stdout);
 temp=fflush(stdout);
 sleep(3);
 temp=ioctl(fileno(stdout),VSETSTATE,plotmd);
 temp=fflush(stdout);
 sleep(3);


/*************************************************************/
/*                                                           */
/*  Output from the top of the plot to the bottom of the     */
/*  plot.                                                    */
/*                                                           */
/*************************************************************/


/*************************************************************/
/*                                                           */
/*  Output from the left edge of the paper to the right      */
/*  edge of the image.  We will do this in a more            */
/*  efficient manner, by checking to see which blocks        */
/*  have and have not been allocated space.  For empty       */
/*  blocks, we can output all zeros.  For allocated          */
/*  blocks, we will output one row from the block all at     */
/*  once, rather than tetching  4 bytes at a time.  Since    */
/*  there can be substantial overhead associated with        */
/*  accessing a dynamic matrix, this should save a           */
```

```
/*  substantial amount of CPU time, and also increase the */
/*  plot speed (up to the limits of the vpc controller).  */
/*                                                        */
/**********************************************************/

    for (x=0; x<max_x ; x++)
       {
        for (y=0;y < max_y ; y=y+COLUMNS_IN_BLOCK)
           if ( INQUIRE_POINTER_TO_BLOCK(image,x,y) == USED)
              temp=fwrite(FETCH_POINTER_TO_BLOCK(image,x,y)
                 + (x%ROWS_IN_BLOCK) * COLUMNS_IN_BLOCK,
                 sizeof(int),COLUMNS_IN_BLOCK,stdout);
           else
              temp=fwrite(dummy,sizeof(int),
                 COLUMNS_IN_BLOCK,stdout);

/**********************************************************/
/*                                                        */
/*  Blank fill the remainder of the scan line.            */
/*                                                        */
/**********************************************************/

        temp=fwrite(dummy,sizeof(int),(MAX_COLUMNS - y),
           stdout);
       };

temp=fflush(stdout);
sleep(3);
temp=ioctl(fileno(stdout),VSETSTATE,prtmd);
temp=fflush(stdout);
sleep(3);
putc(4,stdout);
temp=fflush(stdout);
sleep(3);

return;


}
```

This final routine is where the raster image is actually created. By reducing the number of pages needed to store the raster image, this is where Dynamic Matrices had the most significant effect on the speed of the program. In order to reduce the time spent calculating addresses, this routine makes extensive use of the UPDATE macro, which is functionally equivalent to the C expression X += A, where X would be an element in the dynamic matrix, + is allowed to be any of the standard operations, and A is any expression which lacks side effects. In the case of this routine, the operator is always | (bit wise or). Experience indicates that on a 2 MIP SUN3 with 16 Mbytes of memory, this routine can produce a 400 dot per inch E size raster image for an average E size drawing in a bit over 10 minutes.

```
/*************************************************************/
/*                                                           */
/*  This routine was written by Daniel M. Pressel in the     */
/*  summer of 1989.  It is part of his calcomp to            */
/*  versatec plotting package.                               */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*                                                           */
/*  This software is the property of the United States       */
/*  Government.  When and if it is made available for use    */
/*  by others, it is totally at their own risk.              */
/*  Furthermore, the author is unable to guarantee           */
/*  continued support for this product.  In fact this        */
/*  package has only been installed on systems running       */
/*  rev 3.0.  The installation instructions may only         */
/*  apply to this rev, and probably need modifications if    */
/*  the software is to be successfully installed on          */
/*  future revs.  Any such modifications are the             */
/*  responsibility of the administrators at the sites        */
/*  receiving copies of this software.  Finally, it is       */
/*  illegal to sell either this software, or any             */
/*  derivative software, to other United States              */
/*  Government installations without first informing them    */
/*  that the software is already the property of the         */
/*  United States Government.                                */
/*                                                           */
/*************************************************************/


#include <stdio.h>
#include "dyn.h"
#include "calcomp_defines.h"

void calcomp_raster(delta_x,delta_y,hpstyle,image,offset,
        old_x,old_y,scale,step,thickness,vpstyle)

int delta_x,delta_y;
unsigned int hpstyle;
DYNAMIC_MATRIX *image;
int old_x,old_y;
float scale;
int step;
```

```
int thickness;
unsigned int vpstyle[33];
int offset;

{
 register int level,x,y;

 int bit;
 int column;
 int row;

 float slope;

/*************************************************************/
/*                                                           */
/* Here we are going to rasterize the line to be drawn.      */
/* First we will determine if the absolute value is          */
/* greater than or less than 1.  This allows us to           */
/* maximize the resolution, and to also transparently        */
/* handle both vertical and horizontal lines.  Depending     */
/* upon the value of the slope either X or Y may be          */
/* treated as the independent variable.  Once the            */
/* independent variable has been selected, it will be        */
/* stepped through its range of values at the resolution     */
/* of the plotter.  NOTE:  This program will                 */
/* automatically transpose the X and Y coordinates to        */
/* rotate the drawing to allow A through E size drawings     */
/* to be handled.  This is prefered to using CADDS to        */
/* rotate the drawing, since CADDS has a bug in the          */
/* rotate modifier (at least in REV 3.0 and earlier).        */
/*                                                           */
/*************************************************************/

 if ((0.0 + delta_x)*delta_x > (0.0 + delta_y)*delta_y)
    {

/*************************************************************/
/*                                                           */
/*  Handle lines which are primarily horizontal.             */
/*                                                           */
/*  Note:  Since the lines may be more than one dot wide,    */
/*  horizontal lines, which plot vertically on the paper,    */
/*  may cross word boundaries.  Therefore potentially,       */
/*  two elements of the matrix may need updating.            *
/*                                                           */
/*************************************************************/

     slope=(delta_y+0.0)/delta_x;
     if ( delta_x > 0 )

/*************************************************************/
/*                                                           */
/*  The line goes from left to right.                        */
/*                                                           */
/*************************************************************/
```

```
            {
             for (x=(-step)*offset ; x <= delta_x ;  x+=step )
                 {
                  row=(old_x+x)*scale+WORD_SIZE;
                  column=(slope*x+old_y)*scale-offset+WORD_SIZE;
                  bit=column%WORD_SIZE;
                  column=column/WORD_SIZE;
                  UPDATE_VALUE(image,row,column,
                     hpstyle >> bit , | );
                  column++;
                  UPDATE_VALUE(image,row,column,
                     hpstyle << (WORD_SIZE - bit ), | );
                 }
            }

     else

/*************************************************************/
/*                                                           */
/*  The line goes from right to left.                        */
/*                                                           */
/*************************************************************/

            {
             for (x=step*offset; x >= delta_x ; x-=step )
                 {
                  row=(old_x+x)*scale+WORD_SIZE;
                  column=(slope*x+old_y)*scale-offset+
                     WORD_SIZE;
                  bit=column%WORD_SIZE;
                  column=column/WORD_SIZE;
                  UPDATE_VALUE(image,row,column,
                     hpstyle >> bit , |);
                  column++;
                  UPDATE_VALUE(image,row,column,
                     hpstyle << (WORD_SIZE - bit ), | );
                 };
            }
    }

 else

/*************************************************************/
/*                                                           */
/* The line is primarily vertical.                           */
/*                                                           */
/* Note:  Since the lines may be more than 1 dot wide and    */
/* vertical lines are actually plotted horizontally,         */
/* matrix elements of more than one row may be updated.      */
/*                                                           */
/*************************************************************/

     {
      slope=(delta_x+0.0)/delta_y;
      if ( delta_y > 0 )
```

```
/*************************************************************/
/*                                                           */
/*  The line goes from the bottom to the top.                */
/*                                                           */
/*************************************************************/

          {
           for (y=(-step)*offset ; y <= delta_y ; y+=step )
              {
               row=(slope*y+old_x)*scale-offset+WORD_SIZE;
               column=(old_y+y)*scale+WORD_SIZE;
               bit=column%WORD_SIZE;
               column=column/WORD_SIZE;
               for (level=1 ; level <= thickness ;
                  level++,row++)
                  {
                   UPDATE_VALUE(image,row,column,
                      vpstyle[level] >> bit, | );
                  };
              }
          }

     else

/*************************************************************/
/*                                                           */
/*  The line goes from the top to the bottom.                */
/*                                                           */
/*************************************************************/

          {
           for (y=step*offset; y >= delta_y ; y-=step )
              {
               row=(slope*y+old_x)*scale-offset+WORD_SIZE;
               column=(old_y+y)*scale+WORD_SIZE;
               bit=column%WORD_SIZE;
               column=column/WORD_SIZE;
               for (level=1 ; level <= thickness ;
                  level++,row++)
                 {
                  UPDATE_VALUE(image,row,column,
                     vpstyle[level] >> bit , | );

                 };
              };
          }

    };
 return;
}
```

APPENDIX C: THE DYNAMIC MATRIX MACROS FOR FLOATING POINT DATA

This set of macros has been modified to handle floating point data instead of integer data. Additionally, it contains a large number of new macros, most of which should be considered experimental in nature. In general these additional macros will be most useful when dealing with dense matrices where many if not all of the elements in each sub-matrix will be accessed before moving on to the next sub-matrix. Under these conditions, some of the macros can be simplified (e.g., after storing a value into one of the elements of the sub-matrix, future accesses to that sub-matrix need not check to see if it has already been allocated space). Additionally, versions of many of the macros were created which use a fixed sub-matrix size (at the present time only 8 x 8 element sub-matrices are supported in this way). These macros can be substantially more efficient at performing address calculations since they no longer need to carry out integer division or in most cases integer multiplication. While the small size for the sub-matrices will in some cases provide only a limited benefit in reducing the level of paging, it can be very successful in improving cache utilization for certain types of algorithms.

```
/*************************************************************/
/*                                                           */
/*  This include file was written by Daniel M. Pressel at    */
/*  CRDEC in August of 1988.  It is intended to provide a    */
/*  uniform set of structure definitions and macros for      */
/*  implementing in C his concept of a dynamic matrix.       */
/*  In the fall and winter of 1993 Mr.  Pressel modified     */
/*  the code so that it would work with floating point       */
/*  data instead of integers and added a number of           */
/*  experimental macros.                                     */
/*                                                           */
/*************************************************************/
/*                                                           */
/*  In this file, the following items are defined:           */
/*                                                           */
/*       1)  A scratch variable.  For uniqueness, it is      */
/*           called DYNAMIC_TEMP.                            */
/*                                                           */
/*       2)  Two named constants, USED and UNUSED, which     */
/*           will be used as boolean values in some of       */
/*           the macros to follow.                           */
/*                                                           */
/*       3)  A type definition for the structure used to     */
/*           simulate my concept of dynamic matrices.        */
/*                                                           */
/*       4)  The following macro definitions:                */
/*                                                           */
/*             a)  ALLOCATE:  This macro consists of a       */
/*                 series of seperate statements used to     */
/*                 allocate and initialize the core          */
/*                 structure for a dynamic matrix.           */
/*                                                           */
/*             b)  FETCH_VALUE:  Unlike ALLOCATE, this       */
/*                 macro is composed of a single             */
/*                 arithmetic expression.  As such, it is    */
/*                 suitable for inclusion directly into      */
/*                 an arithmetic expression.  To achieve     */
/*                 this goal, both the comma operator and    */
/*                 the ternary conditional operator were     */
/*                 used.  As such this is an extremely       */
/*                 complicated macro.                        */
```

```
/*                                                                  */
/*          c)  FETCH_POINTER_TO_BLOCK:  This macro is              */
/*              also designed to be directly included               */
/*              into arithimetic expressions and                    */
/*              function calls.  Given a row and                    */
/*              column number, this macro will                      */
/*              determine in which of the blocks the                */
/*              matrix was subdivided into, the                     */
/*              element resides, it then returns a                  */
/*              pointer to the block of memory                      */
/*              containing this element.  NOTE:  If                 */
/*              the specified block has not had any                 */
/*              values stored into it, then the NULL                */
/*              pointer is returned.  It is therefore               */
/*              imperative that the                                 */
/*              INQUIRE_POINTER_TO_BLOCK macro be used              */
/*              first to see if the block exists.                   */
/*                                                                  */
/*          d)  INQUIRE_POINTER_TO_BLOCK:  This macro               */
/*              returns a boolean value of either USED              */
/*              or UNUSED, depending upon if anything               */
/*              other than the default value is stored              */
/*              in the elements of the specified                    */
/*              block.  It may be used in an                        */
/*              expression, but probably is best used               */
/*              as part of if, while, and for                       */
/*              statements.                                         */
/*                                                                  */
/*          e)  STORE_VALUE:  Here is another macro,                */
/*              which is composed of multiple                       */
/*              statements, and must not be included                */
/*              in any expressions.  It will update                 */
/*              the value of an element in a dynamic                */
/*              matrix, allocating space for a block                */
/*              as needed.  When a block is allocated,              */
/*              this macro will also initialize each                */
/*              of its elements.  Since this takes                  */
/*              some time, and increases the amount of              */
/*              memory in use,  this macro will                     */
/*              ignore requests to store the default                */
/*              value into an element of the matrix if              */
/*              it resides in a previously unused                   */
/*              block.                                              */
/*                                                                  */
/*          f)  DEALLOCATE:  This is a multi statement              */
/*              macro used to free all of the memory                */
/*              previously used by a dynamic matrix.                */
/*                                                                  */
/*          g)  UPDATE_VALUE:  This combines the                    */
/*              functionallity of FETCH_VALUE with                  */
/*              STORE_VALUE.  There are two added                   */
/*              arguments.  The second extra argument               */
/*              contains a binary operator which will               */
/*              be applied to value stored in the                   */
/*              dyanmic matrix.  The first additional               */
/*              argument contains a value or                        */
```

```
/*                  expression which will come to the      */
/*                  right of the binary operator.  This    */
/*                  macro is not required, but its use     */
/*                  may result in faster running code.     */
/*                                                         */
/*                                                         */
/***********************************************************/

/***********************************************************/
/*                                                         */
/*  NOTE:  As with all macros, one should never allow      */
/*  any of the arguments to the macros to have side        */
/*  effects.  If side effects are present, they will       */
/*  almost always take effect several times, with usually  */
/*  undesirable consequences.  Additionally, one may       */
/*  include expressions as part of most of the arguments,  */
/*  but care must be taken in doing so.  If the            */
/*  expressions are computationally intensive, or call     */
/*  functions they may have severe negative speed          */
/*  consequences.  If this becomes a problem, the use of   */
/*  temporary or scratch variables would be well           */
/*  advised.                                               */
/*                                                         */
/***********************************************************/

/***********************************************************/
/*                                                         */
/*  Several additional macros have been added to this      */
/*  package.  Those with _8_8 appended to the name use a   */
/*  fixed submatrix size of 8 x 8 elements.  This allows   */
/*  all integer division and multiplication to be          */
/*  replaced with bit shifting, while the mod operator     */
/*  was replaced with bit masks.  This makes these macros  */
/*  dramatically more efficient on most hardware.          */
/*  Additionally, some macros with the  designation PRE    */
/*  were created to allow for the more efficient           */
/*  allocation and manipulation of dense matrices.         */
/*  These macros must only be used with Dynamic Matrices   */
/*  initialized using the PREALLOCATE macro.  Dynamic      */
/*  Matrices so initialized may still be manipulated       */
/*  using non PRE macros, but this will be less            */
/*  efficient.  Finally, there are a large number of new   */
/*  macros which are designed to allow for the more        */
/*  efficient manipulation of Dynamic Matrices under       */
/*  specific conditions.  In general the  utility of       */
/*  these macros has not been established, and they        */
/*  should be considered to be unsupported.                */
/*                                                         */
/***********************************************************/

/***********************************************************/
/*                                                         */
/*  This software is the property of the United States     */
/*  Government.  When and if it is made available for use  */
/*  by others, it is totally at their own risk.            */
/*  Furthermore, the author is unable to guarantee         *
```

```
                                                                    */
/*  continued support for this product.  Any                        */
/*  modifications are the responsibility of the users.              */
/*  Finally, it is illegal to sell either this software,            */
/*  or any derivative software, to other United States              */
/*  Government installations without first informing them           */
/*  that the software is already the property of the                */
/*  United States Government.                                       */
/*                                                                  */
/********************************************************************/


void *malloc();
void *calloc();
float  *DYNAMIC_TEMP;
float   DYNAMIC_TEMP_VALUE;
float  *DYNAMIC_LAST_ADDRESS;

#define USED 1
#define UNUSED 0

typedef struct
   {
    float initial_value;
    int number_of_rows;
    int number_of_columns;
    float **pointer_to_matrix;
    int rows_in_block;
    int columns_in_block;
    int blocks_in_row;
   }   DYNAMIC_MATRIX;

#define ALLOCATE(pointer,init_val,num_of_rows,\
           num_of_columns,num_rows_in_block,\
           num_columns_in_block) \
        pointer=(DYNAMIC_MATRIX *)\
           malloc(sizeof(DYNAMIC_MATRIX));\
        pointer -> initial_value=init_val;\
        pointer -> number_of_rows=num_of_rows;\
        pointer -> number_of_columns=num_of_columns;\
        pointer -> blocks_in_row=(num_of_columns +\
           num_columns_in_block -1 )/num_columns_in_block;\
        pointer -> pointer_to_matrix=(float **)\
           calloc(((num_of_rows + num_rows_in_block - 1)/\
           num_rows_in_block)*(pointer -> blocks_in_row),\
           sizeof(float *));\
        pointer -> rows_in_block=num_rows_in_block;\
        pointer -> columns_in_block=num_columns_in_block;

#define FETCH_VALUE(pointer,row_number,column_number) \
        ((DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
           (row_number)/(pointer -> rows_in_block) *\
           (pointer -> blocks_in_row) + (column_number)/\
           (pointer -> columns_in_block))),\
           ((DYNAMIC_TEMP != 0) ? *((DYNAMIC_TEMP) +\
           (row_number) % (pointer -> rows_in_block) *\
```

```
            (pointer -> columns_in_block ) +\
            (column_number) % (pointer -> columns_in_block))\
            : (pointer -> initial_value)))

#define FETCH_POINTER_TO_BLOCK(pointer,row_number,\
            column_number) \
         (*(pointer -> pointer_to_matrix + (row_number) \
            / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)/\
            (pointer -> columns_in_block)))


#define INQUIRE_POINTER_TO_BLOCK(pointer,row_number,\
            column_number) \
         ((*(pointer -> pointer_to_matrix + (row_number) \
            / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)/\
            (pointer -> columns_in_block))) != NULL\
            ? USED : UNUSED )

#define STORE_VALUE(pointer,row_number,column_number,\
            value) \
         DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
            (row_number) / (pointer -> rows_in_block) *\
            (pointer -> blocks_in_row) + (column_number)\
            / (pointer -> columns_in_block)); \
         if (DYNAMIC_TEMP != 0) \
            *((DYNAMIC_TEMP) + (row_number) %\
            (pointer -> rows_in_block) * \
            (pointer -> columns_in_block ) +\
            (column_number) % (pointer -> columns_in_block))\
            = (value) ; \
         \
         else \
            if ( (value) !=  (pointer -> initial_value))\
               {\
                register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
                register float DYNAMIC_INITIAL=\
                   (pointer -> initial_value);\
                if (DYNAMIC_INITIAL == 0.0)\
                   DYNAMIC_TEMP=(float *)\
                      calloc((pointer -> rows_in_block) *\
                      (pointer -> columns_in_block),\
                      sizeof(float));\
                else\
                   {\
                    DYNAMIC_TEMP=(float *)
                       malloc((pointer -> rows_in_block) * \
                       (pointer -> columns_in_block) *\
                       sizeof(float));\
                    *(pointer -> pointer_to_matrix +\
                       (row_number)/\
                       (pointer -> rows_in_block) *\
                       (pointer -> blocks_in_row) \
                       + (column_number)/\
                       (pointer -> columns_in_block)) \
```

```
                    =DYNAMIC_TEMP; \
                  \
                  for(DYNAMIC_ROW=0; DYNAMIC_ROW <\
                     (pointer -> rows_in_block) ;\
                     DYNAMIC_ROW++)\
                     for(DYNAMIC_COLUMN=0;\
                        DYNAMIC_COLUMN < \
                        (pointer -> columns_in_block) ;\
                        DYNAMIC_COLUMN++) \
                        \
                        *(DYNAMIC_TEMP + DYNAMIC_ROW * \
                           (pointer -> columns_in_block) \
                           + DYNAMIC_COLUMN) =\
                           DYNAMIC_INITIAL;\
                  };\
               *(pointer -> pointer_to_matrix +\
                  (row_number)/(pointer -> rows_in_block)\
                  * (pointer -> blocks_in_row) \
                  + (column_number)/\
                  (pointer -> columns_in_block)) \
                  = DYNAMIC_TEMP; \
               *((DYNAMIC_TEMP) + (row_number) %\
                  (pointer -> rows_in_block)\
                  * (pointer -> columns_in_block ) +\
                  (column_number) % \
                  (pointer -> columns_in_block))=(value);\
            };;

#define DEALLOCATE(pointer) \
         { \
          register int column,row; \
          for ( row=0 ; row <\
             (pointer -> number_of_rows + \
             pointer -> rows_in_block - 2); \
             row+=(pointer -> rows_in_block)) \
             \
             for ( column=0 ; column <\
                (pointer -> number_of_columns + \
                pointer -> columns_in_block - 2); \
                column+=(pointer -> columns_in_block))\
                \
                free(*((pointer -> pointer_to_matrix)++));\
         };
#define UPDATE_VALUE(pointer,row_number,column_number,\
           value,operation) \
        DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
           (row_number) / (pointer -> rows_in_block) *\
           (pointer -> blocks_in_row) + (column_number)\
           / (pointer -> columns_in_block)); \
        if (DYNAMIC_TEMP != 0) \
             DYNAMIC_TEMP_VALUE = (*((DYNAMIC_TEMP) +\
                (row_number) % (pointer -> rows_in_block)\
                * (pointer -> columns_in_block ) \
                + (column_number) %\
                (pointer -> columns_in_block))) \
                operation (value) ; \
```

```
\
else \
     DYNAMIC_TEMP_VALUE =\
        (pointer -> initial_value) operation\
        (value);\
\
if (DYNAMIC_TEMP != 0 ) \
   *((DYNAMIC_TEMP) + (row_number) %\
      (pointer -> rows_in_block) * \
      (pointer -> columns_in_block ) +\
      (column_number) % \
      (pointer -> columns_in_block)) =\
      DYNAMIC_TEMP_VALUE ; \
\
else \
   if ( DYNAMIC_TEMP_VALUE !=\
      (pointer -> initial_value))\
      {\
       register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
       register float DYNAMIC_INITIAL=\
          (pointer -> initial_value);\
       if (DYNAMIC_INITIAL == 0.0)\
          DYNAMIC_TEMP=(float *)\
             calloc((pointer -> rows_in_block) *\
             (pointer -> columns_in_block),\
             sizeof(float));\
       else\
          {\
           DYNAMIC_TEMP=(float *)\
              malloc((pointer -> rows_in_block) *\
              (pointer -> columns_in_block) *\
              sizeof(float));\
           *(pointer -> pointer_to_matrix +\
              (row_number)/\
              (pointer -> rows_in_block) *\
              (pointer -> blocks_in_row) \
              + (column_number)/\
              (pointer -> columns_in_block)) \
              =DYNAMIC_TEMP; \
           \
          for(DYNAMIC_ROW=0; DYNAMIC_ROW <\
             (pointer -> rows_in_block) ;\
             DYNAMIC_ROW++)\
             for(DYNAMIC_COLUMN=0; DYNAMIC_COLUMN\
                < (pointer -> columns_in_block) ;\
                DYNAMIC_COLUMN++) \
                \
                *(DYNAMIC_TEMP + DYNAMIC_ROW * \
                   (pointer -> columns_in_block) \
                   + DYNAMIC_COLUMN) =\
                   DYNAMIC_INITIAL;\
          };\
         *(pointer -> pointer_to_matrix +\
             (row_number) \
             / (pointer -> rows_in_block) *\
             (pointer -> blocks_in_row) \
```

```
                + (column_number)/\
                (pointer -> columns_in_block)) \
                = DYNAMIC_TEMP; \
             *((DYNAMIC_TEMP) + (row_number) %\
                (pointer -> rows_in_block)\
                * (pointer -> columns_in_block ) +\
                (column_number) % \
                (pointer -> columns_in_block)) =\
                DYNAMIC_TEMP_VALUE ; \
          };;

#define DENSE_STORE_VALUE(pointer,row_number,\
      column_number,value) \
     DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
      (row_number)\
      / (pointer -> rows_in_block) *\
      (pointer -> blocks_in_row) + (column_number)\
      / (pointer -> columns_in_block)); \
     if (DYNAMIC_TEMP != 0) \
      *((DYNAMIC_TEMP) + (row_number) %\
         (pointer -> rows_in_block) * \
         (pointer -> columns_in_block ) +\
         (column_number) % \
         (pointer -> columns_in_block)) =\
         (value) ; \
     \
     else \
      if ( (value) !=\
         (pointer -> initial_value))\
         {\
          DYNAMIC_TEMP=(float *)\
             malloc((pointer -> rows_in_block) * \
             (pointer -> columns_in_block) *\
             sizeof(float));\
          *(pointer -> pointer_to_matrix +\
             (row_number)/(pointer -> rows_in_block)\
             * (pointer -> blocks_in_row) \
             + (column_number)/ \
             (pointer -> columns_in_block)) \
             =DYNAMIC_TEMP; \
          \
          *((DYNAMIC_TEMP) + (row_number) % \
             (pointer -> rows_in_block)\
             * (pointer -> columns_in_block ) +\
             (column_number) % \
             (pointer -> columns_in_block)) =\
             (value) ; \
          };;

#define FAST_STORE_VALUE(pointer,row_number,\
      column_number,value) \
     DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
      (row_number)/(pointer -> rows_in_block)\
      * (pointer -> blocks_in_row) \
      + (column_number) /\
      (pointer -> columns_in_block)); \
```

```
            *((DYNAMIC_TEMP) + (row_number) %\
               (pointer -> rows_in_block) * \
               (pointer -> columns_in_block ) +\
               (column_number) %\
               (pointer -> columns_in_block)) = (value) ;;

#define PRE_SERIAL_STORE_VALUE(pointer,row_number,\
               column_number,value) \
            DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
               (row_number)/(pointer -> rows_in_block) *\
               (pointer -> blocks_in_row) + (column_number)\
               / (pointer -> columns_in_block)); \
            if (DYNAMIC_TEMP != 0) \
               *((DYNAMIC_TEMP) + (row_number) %\
               (pointer -> rows_in_block) * \
               (pointer -> columns_in_block ) +\
               (column_number) % \
               (pointer -> columns_in_block)) = (value) ; \
            \
            else \
               if ( (value) !=  (pointer -> initial_value))\
                  {\
                   register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
                   register float DYNAMIC_INITIAL=\
                      (pointer -> initial_value);\
                   if (DYNAMIC_INITIAL == 0.0)\
                      DYNAMIC_TEMP=(float *)\
                         calloc((pointer -> rows_in_block) * \
                         (pointer -> columns_in_block),\
                         sizeof(float));\
            else\
               {\
                DYNAMIC_TEMP=(float *)\
                   malloc((pointer -> rows_in_block) * \
                   (pointer -> columns_in_block) *\
                   sizeof(float));\
                *(pointer -> pointer_to_matrix +\
                   (row_number)/\
                   (pointer -> rows_in_block) *\
                   (pointer -> blocks_in_row) \
                   + (column_number)/\
                   (pointer -> columns_in_block)) \
                   =DYNAMIC_TEMP; \
               \
               for(DYNAMIC_ROW=0; DYNAMIC_ROW <\
                  (pointer -> rows_in_block) ;\
                  DYNAMIC_ROW++)\
                  for(DYNAMIC_COLUMN=0;\
                     DYNAMIC_COLUMN < \
                     (pointer -> columns_in_block) ;\
                     DYNAMIC_COLUMN++) \
                     \
                     *(DYNAMIC_TEMP + DYNAMIC_ROW * \
                        (pointer -> columns_in_block) \
                        + DYNAMIC_COLUMN) =\
                        DYNAMIC_INITIAL;\
```

```
                    };\
                 *(pointer -> pointer_to_matrix +\
                    (row_number)/(pointer -> rows_in_block)\
                    * (pointer -> blocks_in_row) \
                    + (column_number)/\
                    (pointer -> columns_in_block)) \
                    = DYNAMIC_TEMP; \
                 *((DYNAMIC_TEMP) + (row_number) %\
                    (pointer -> rows_in_block)\
                    * (pointer -> columns_in_block ) +\
                    (column_number) % \
                    (pointer -> columns_in_block)) =\
                    (value) ; \
                 };\
              DYNAMIC_LAST_ADDRESS = ((DYNAMIC_TEMP) +\
                 (row_number) % (pointer -> rows_in_block) *\
                 (pointer -> columns_in_block )\
                 + (column_number) %\
                 (pointer -> columns_in_block));;

#define SERIAL_STORE_VALUE(value) \
           *(++DYNAMIC_LAST_ADDRESS) = (value) ;;

#define FETCH_VALUE_8_8(pointer,row_number,column_number) \
        ((DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
             (row_number >> 3) * (pointer -> blocks_in_row)\
             + (column_number >> 3))), \
             ((DYNAMIC_TEMP != 0) ? *((DYNAMIC_TEMP) +\
             ((row_number & 0X07) << 3) +\
             (column_number & 0X07)) :\
             (pointer -> initial_value)))

#define FETCH_POINTER_TO_BLOCK_8_8(pointer,row_number,\
           column_number) \
        (*(pointer -> pointer_to_matrix +\
           (row_number >> 3) * (pointer -> blocks_in_row)\
           + (column_number >> 3)))

#define STORE_VALUE_8_8(pointer,row_number,column_number,\
           value) \
           DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
              (row_number >> 3) * (pointer -> blocks_in_row)\
              + (column_number >> 3)); \
           if (DYNAMIC_TEMP != 0) \
              *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
                 + (column_number & 0X07)) = (value) ; \
           \
           else \
              if ( (value) !=  (pointer -> initial_value))\
                 {\
                  register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
                  register float DYNAMIC_INITIAL=\
                     (pointer -> initial_value);\
                  if (DYNAMIC_INITIAL == 0.0)\
                     DYNAMIC_TEMP=(float *)\
                        calloc(64,sizeof(float));\
```

```
                    else\
                       {\
                        DYNAMIC_TEMP=(float *)\
                           malloc(64 * sizeof(float));\
                        *(pointer -> pointer_to_matrix +\
                           (row_number >> 3)/ \
                           (pointer -> blocks_in_row) +\
                           (column_number >> 3))=DYNAMIC_TEMP;\
                        \
                       for(DYNAMIC_ROW=0; DYNAMIC_ROW < 8 ;\
                          DYNAMIC_ROW++)\
                          for(DYNAMIC_COLUMN=0; DYNAMIC_COLUMN\
                             < 8 ; DYNAMIC_COLUMN++) \
                             \
                             *(DYNAMIC_TEMP +\
                                (DYNAMIC_ROW << 3) \
                                + DYNAMIC_COLUMN) =\
                                DYNAMIC_INITIAL;\
                       };\
                    *(pointer -> pointer_to_matrix +\
                       (row_number >> 3) \
                       * (pointer -> blocks_in_row) +\
                       (column_number >> 3)) = DYNAMIC_TEMP; \
                    *((DYNAMIC_TEMP) +\
                       ((row_number & 0X07) << 3)\
                       + (column_number & 0X07)) = (value) ; \
                    };;


#define DENSE_STORE_VALUE_8_8(pointer,row_number,\
             column_number,value) \
          DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
             (row_number >> 3) * (pointer -> blocks_in_row)\
             + (column_number >> 3)); \
          if (DYNAMIC_TEMP != 0) \
             *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
                + (column_number & 0X07)) = (value) ; \
          \
          else \
             if ( (value) !=  (pointer -> initial_value))\
                {\
                 DYNAMIC_TEMP=(float *)\
                    malloc(64 * sizeof(float));\
                 *(pointer -> pointer_to_matrix +\
                    (row_number >> 3) \
                    * (pointer -> blocks_in_row) +\
                    (column_number >> 3))=DYNAMIC_TEMP; \
                 \
                 *((DYNAMIC_TEMP) +\
                    ((row_number & 0X07) << 3) \
                       + (column_number & 0X07)) = (value) ; \
                    };;

     #define PRE_SERIAL_STORE_VALUE_8_8(pointer,row_number,\
                column_number,value) \
             DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
```

```
      (row_number >> 3) * (pointer -> blocks_in_row)\
      + (column_number >> 3)); \
   if (DYNAMIC_TEMP != 0) \
      *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
         + (column_number & 0X07)) = (value) ; \
   \
   else \
      if ( (value) !=  (pointer -> initial_value))\
         {\
          register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
          register float DYNAMIC_INITIAL=\
             (pointer -> initial_value);\
          if (DYNAMIC_INITIAL == 0.0)\
             DYNAMIC_TEMP=(float *)\
                calloc(64,sizeof(float));\
          else\
             {\
              DYNAMIC_TEMP=(float *)\
                 malloc(64 * sizeof(float));\
              *(pointer -> pointer_to_matrix +\
                 (row_number >> 3) \
                 * (pointer -> blocks_in_row) +\
                 (column_number >> 3))=DYNAMIC_TEMP; \
              \
             for(DYNAMIC_ROW=0; DYNAMIC_ROW < 8 ;\
                DYNAMIC_ROW++)\
                for(DYNAMIC_COLUMN=0;\
                   DYNAMIC_COLUMN < 8; \
                   DYNAMIC_COLUMN++) \
                   \
                   *(DYNAMIC_TEMP +\
                      (DYNAMIC_ROW << 3) \
                      + DYNAMIC_COLUMN) =\
                      DYNAMIC_INITIAL;\
             };\
          *(pointer -> pointer_to_matrix +\
             (row_number >> 3) \
             * (pointer -> blocks_in_row) +\
             (column_number >> 3)) = DYNAMIC_TEMP; \
          *((DYNAMIC_TEMP) +\
             ((row_number & 0X07) << 3) \
             + (column_number & 0X07)) = (value) ; \
         };\
      DYNAMIC_LAST_ADDRESS = ((DYNAMIC_TEMP) +\
         ((row_number & 0X07) << 3) \
         + (column_number & 0X07));;

#define PRE_SERIAL_UPDATE_VALUE_8_8(pointer,row_number,\
      column_number,value,operation) \
   DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
      (row_number >> 3) * (pointer -> blocks_in_row)\
      + (column_number >> 3)); \
   if (DYNAMIC_TEMP != 0) \
      DYNAMIC_TEMP_VALUE = (*((DYNAMIC_TEMP) +\
         ((row_number & 0X07) << 3) +\
         (column_number & 0X07))) operation\
         (value) ; \
```

```
          \
          else \
               DYNAMIC_TEMP_VALUE =\
          (pointer -> initial_value) operation\
          (value);\
\
if (DYNAMIC_TEMP != 0 ) \
   *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
      + (column_number & 0X07)) =\
      DYNAMIC_TEMP_VALUE ; \
\
else \
   if ( DYNAMIC_TEMP_VALUE !=\
      (pointer -> initial_value))\
      {\
       register int DYNAMIC_COLUMN, DYNAMIC_ROW;\
       register float DYNAMIC_INITIAL=\
          (pointer -> initial_value);\
       if (DYNAMIC_INITIAL == 0.0)\
          DYNAMIC_TEMP=(float *)\
             calloc(64,sizeof(float));\
       else\
          {\
           DYNAMIC_TEMP=(float *)\
              malloc(64 * sizeof(float));\
           *(pointer -> pointer_to_matrix +\
              (row_number >> 3) \
              * (pointer -> blocks_in_row) +\
              (column_number >> 3))=DYNAMIC_TEMP;\
           \
          for(DYNAMIC_ROW=0; DYNAMIC_ROW < 8 ;\
             DYNAMIC_ROW++)\
             for(DYNAMIC_COLUMN=0;\
                DYNAMIC_COLUMN < 8 ; \
                DYNAMIC_COLUMN++) \
                \
                *(DYNAMIC_TEMP +\
                   (DYNAMIC_ROW << 3) \
                   + DYNAMIC_COLUMN) =\
                   DYNAMIC_INITIAL;\
          };\
       *(pointer -> pointer_to_matrix +\
          (row_number >> 3) \
          * (pointer -> blocks_in_row) +\
          (column_number >> 3)) = DYNAMIC_TEMP; \
       *((DYNAMIC_TEMP) +\
          ((row_number & 0X07) << 3)\
          + (column_number & 0X07)) =\
          DYNAMIC_TEMP_VALUE ; \
      };;\
   DYNAMIC_LAST_ADDRESS = ((DYNAMIC_TEMP) +\
      ((row_number & 0X07) << 3) \
      + (column_number & 0X07));;
```

```
#define SERIAL_UPDATE_VALUE(value,operation) \
          *(++DYNAMIC_LAST_ADDRESS) =\
             *DYNAMIC_LAST_ADDRESS operation (value) ;;

#define OFFSET_STORE_VALUE(value,offset) \
          *(DYNAMIC_LAST_ADDRESS + (offset)) = (value) ;;

#define PREALLOCATE_8_8(pointer,num_of_rows,\
          num_of_columns) \
        {\
         register int DYNAMIC_COLUMN,\
            DYNAMIC_ROW;\
         pointer=(DYNAMIC_MATRIX *)\
            malloc(sizeof(DYNAMIC_MATRIX));\
         pointer -> initial_value=0.0;\
         pointer -> number_of_rows=num_of_rows;\
         pointer -> number_of_columns=num_of_columns;\
         pointer -> blocks_in_row=\
            ((num_of_columns + 7) >> 3) + 1;\
         pointer -> pointer_to_matrix=(float **)\
            calloc((((num_of_rows + 7) >> 3) + 1)*\
            (pointer ->  blocks_in_row),sizeof(float *));\
         pointer -> rows_in_block=8;\
         pointer -> columns_in_block=8;\
         DYNAMIC_TEMP=(float *)\
            calloc((((num_of_columns + 7) >> 3) << 3)\
            + 8) * ((((num_of_rows + 7) >> 3) << 3) + 8),\
            sizeof(float));\
         for (DYNAMIC_ROW=0; DYNAMIC_ROW <=\
            ((num_of_rows +7) >> 3); DYNAMIC_ROW++)\
            for (DYNAMIC_COLUMN=0; DYNAMIC_COLUMN <=\
               ((num_of_columns + 7) >> 3);\
               DYNAMIC_COLUMN++)\
               \
               {\
                *(pointer -> pointer_to_matrix +\
                   pointer -> blocks_in_row *\
                   DYNAMIC_ROW + DYNAMIC_COLUMN) =\
                   DYNAMIC_TEMP;\
                DYNAMIC_TEMP += 64;\
               };\
        };

#define FAST_STORE_VALUE_8_8(pointer,row_number,\
          column_number,value) \
        DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
          (row_number >> 3) * (pointer -> blocks_in_row)\
          + (column_number >> 3)); \
        *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
          + (column_number & 0X07)) = (value) ;

#define FAST_PRE_SERIAL_STORE_VALUE_8_8(pointer,\
          row_number,column_number,value) \
        DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
          (row_number >> 3) * (pointer -> blocks_in_row)\
          + (column_number >> 3)); \
```

```
        *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3) \
           + (column_number & 0X07)) = (value) ; \
        DYNAMIC_LAST_ADDRESS = ((DYNAMIC_TEMP) +\
           ((row_number & 0X07) << 3) \
           + (column_number & 0X07));;

#define FAST_PRE_SERIAL_UPDATE_VALUE_8_8(pointer,\
           row_number,column_number,value,operation) \
        DYNAMIC_TEMP= *(pointer -> pointer_to_matrix +\
           (row_number >> 3) * (pointer -> blocks_in_row)\
           + (column_number >> 3)); \
        DYNAMIC_TEMP_VALUE = (*((DYNAMIC_TEMP) +\
           ((row_number & 0X07) << 3) \
           + (column_number & 0X07))) operation (value);\
        *((DYNAMIC_TEMP) + ((row_number & 0X07) << 3)\
           + (column_number & 0X07))=DYNAMIC_TEMP_VALUE;\
        DYNAMIC_LAST_ADDRESS = ((DYNAMIC_TEMP) +\
           ((row_number & 0X07) << 3) \
           + (column_number & 0X07));;

#define OFFSET_UPDATE_VALUE(value,offset,operation) \
           *(DYNAMIC_LAST_ADDRESS + (offset)) =\
           *(DYNAMIC_LAST_ADDRESS + (offset)) \
              operation (value) ;;
```

INTENTIONALLY LEFT BLANK.

BIBLIOGRAPHY

Bach, M. J. The Design of the UNIX Operating System. Englewood Cliffs, NJ: Prentice Hall, 1986.

Deitel, H. M. An Introduction to Operating Systems. Reading, MA: Addison-Wesley Publishing Co., 1984.

Hayes, J. P. Computer Architecture and Organization 2nd edition. New York, NY: McGraw-Hill Book Co., 1988.

Horowitz, E. Fundamentals of Computer Algorithms. Rockville, MD: Computer Science Press, 1978.

Kane, G. MIPS RISC Architecture. Englewood Cliffs, NJ: Prentice Hall, 1992.

Kurzban, S. A. Operating Systems Principles 2nd edition. New York, NY: Van Nostrand Reinhold Co., 1984.

Leffler, S. J. The Design and Implementation of the 4.3BSD UNIX Operating System. Reading, MA: Addison-Wesley Publishing Company, 1989.

Leonard, T. E. (ed.). VAX Architecture Reference Manual. Digital Equipment Corp., Bedford, MA, 1987.

Nemeth, E. UNIX System Administration Handbook. Englewood Cliffs, NJ: Prentice Hall, 1989.

Prasad, N. S. IBM Mainframes Architecture and Design. New York, NY: McGraw-Hill Book Co., 1989.

Sedgewick, R. Algorithms. Reading, MA: Addison-Wesley Publishing Co., 1983.

Stone, H. High-Performance Computer Architecture. Reading, MA: Addison-Wesley Publishing Co., 1987.

Tenenbaum, A. M. Data Structures Using Pascal. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Waite, M. UNIX Papers for UNIX Developers and Power Users. Indianapolis, IN: Howard W. Sams & Co., 1987.

INTENTIONALLY LEFT BLANK.

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER ATTN: DTIC-DDA CAMERON STATION ALEXANDRIA VA 22304-6145
1	COMMANDER US ARMY MATERIEL COMMAND ATTN: AMCAM 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN: AMSRL-OP-SD-TA/ RECORDS MANAGEMENT 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR US ARMY RESEARCH LABORATORY ATTN: AMSRL-OP-SD-TL/ TECHNICAL LIBRARY 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN: AMSRL-OP-SD-TP/ TECH PUBLISHING BRANCH 2800 POWDER MILL RD ADELPHI MD 20783-1145
2	COMMANDER US ARMY ARDEC ATTN: SMCAR-TDC PICATINNY ARSENAL NJ 07806-5000
1	DIRECTOR BENET LABORATORIES ATTN: SMCAR-CCB-TL WATERVLIET NY 12189-4050
1	DIRECTOR US ARMY ADVANCED SYSTEMS RESEARCH AND ANALYSIS OFFICE ATTN: AMSAT-R-NR/MS 219-1 AMES RESEARCH CENTER MOFFETT FIELD CA 94035-1000

NO. OF COPIES	ORGANIZATION
1	COMMANDER US ARMY MISSILE COMMAND ATTN: AMSMI-RD-CS-R (DOC) REDSTONE ARSENAL AL 35898-5010
1	COMMANDER US ARMY TANK-AUTOMOTIVE COMMAND ATTN: AMSTA-JSK (ARMOR ENG BR) WARREN MI 48397-5000
1	DIRECTOR US ARMY TRADOC ANALYSIS COMMAND ATTN: ATRC-WSR WSMR NM 88002-5502
1	COMMANDANT US ARMY INFANTRY SCHOOL ATTN: ATSH-WCB-O FORT BENNING GA 31905-5000
	ABERDEEN PROVING GROUND
2	DIR, USAMSAA ATTN: AMXSY-D AMXSY-MP/H COHEN
1	CDR, USATECOM ATTN: AMSTE-TC
1	DIR, USAERDEC ATTN: SCBRD-RT
1	CDR, USACBDCOM ATTN: AMSCB-CII
1	DIR, USARL ATTN: AMSRL-SL-I
5	DIR, USARL ATTN: AMSRL-OP-AP-L

NO. OF
COPIES ORGANIZATION

25 DIR, USARL
ATTN: AMSRL-CI/WILLIAM H MERMAGEN
AMSRL-CI-C/WALTER B STUREK
AMSRL-CI-CA/
B D BROOME
A K CELMINS
J C DUMER
T A KORJACK
T P HANRATTY
R A HELFMAN
N R PATEL
D M PRESSEL
C K ZOLTANI
AMSRL-CI-CC/
B L REICHARD
A E M BRODEEN
F S BRUNDICK
H CATON
S C CHAMBERLAIN
A B COOPER III
A R DOWNS
D A GWYN
G W HARTWIG JR
R C KASTE
M C LOPEZ
L F WRENCHER
S D KOTHENBEUTEL
AMSRL-CI-CD/J D GANTT

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-MR-198 Date of Report November 1994

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001, APG, MD

Postage will be paid by addressee

Director
U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066